2003

Enhanced PL-WAP tree method for incremental mining of sequential patterns.

Min Chen
University of Windsor

Enhanced PL-WAP Tree Method for Incremental Mining of Sequential Patterns

By

Min Chen

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science in
Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

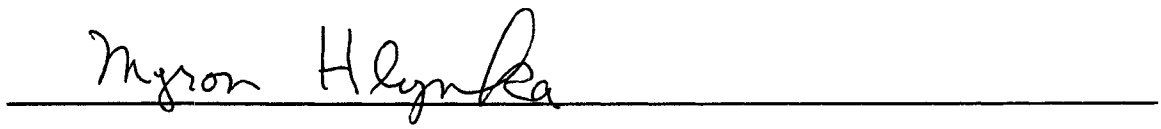
Windsor, Ontario, Canada

2003

Your file Votre référence
ISBN: 0-612-84552-4
Our file Notre référence
ISBN: 0-612-84552-4

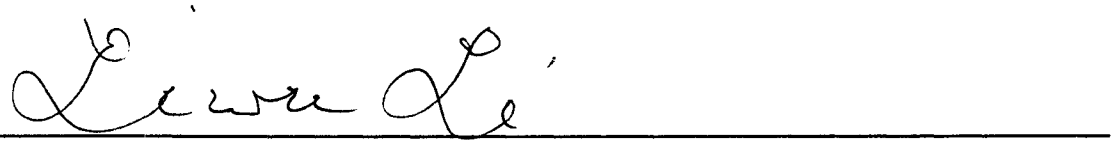
Min Chen 2003

APPROVED BY



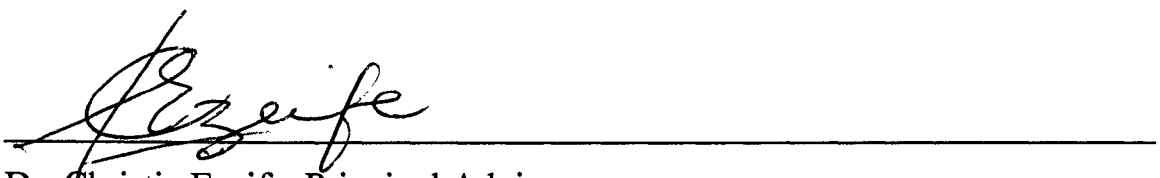
Dr. Myron Hlynka, External Reader

Department of Mathematics and Statistics



Dr. Liwu Li, Internal Reader

School of Computer Science



Dr. Christie Ezeife, Principal Advisor

School of Computer Science

Dr. Xiao Jun Chen, Chair of defense

School of Computer Science

Abstract

As the popularity of the world wide web explodes, massive amounts of data are gathered by web servers in the form of web access logs, which can be used to find web users' surf patterns. A surf pattern on the web is a sequence of web page accesses. Sequential mining as web usage mining has been used in improving web site design, increasing volume of e-business and providing marketing decision support.

Since data streams flow into databases every day, old patterns may be stale and need to be updated. The most straightforward approach for updating generated sequential patterns is running an existing algorithm on the entire updated database (old and new data) from scratch without reusing previously mined frequent sequences. Algorithms for mining sequential patterns from scratch include WAP, PLWAP and apriori-based GSP. Obviously, this approach is not efficient and is time consuming because it has to repeat many computations previously done. Few algorithms developed for incremental mining of sequential patterns are apriori-based and suffer from draw backs of: 1) generating numerous candidate itemsets at each level and 2) scanning the incremental part of the database at each level and scanning the whole database many times when some previous small itemsets become large in the incremental part of the database.

This thesis proposes PL4UP and EPL4UP algorithms which use the PLWAP tree structure to incrementally update sequential patterns. PL4UP does not scan old DB except when previous small 1-itemsets become large in updated database during which time its scans only all transactions in the old database that contain any small itemsets. EPL4UP rebuilds the old PLWAP tree using only the list of previous small itemsets once rather than scanning the entire old database twice like original PLWAP. PL4UP and EPL4UP first update old frequent patterns on the small PLWAP tree built for only the incremented part of the database, then they compare new added patterns generated from the small tree with the old frequent patterns to reduce the number of patterns to be checked on the old PLWAP tree. Thus, the new proposed algorithms PL4UP and EPL4UP achieve better performance than existing incremental sequential mining algorithms because of reduced scan of original database and no generation of candidate itemsets.

Key Words:

Data mining, web usage mining, association rule, sequential mining, patterns, apriori algorithm, WAP tree, PLWAP tree, PL4UP, EPL4UP

To my wife

Acknowledgements

I would like to express my appreciation to the people who gave me help in writing this thesis.

I would like to deeply thank my supervisor Dr. Ezeife for her invaluable comments, encouragement, instructions, and support during the thesis work and all my graduate studies. Without her help, this thesis would not have been completed.

I would also thank my internal reader Dr. Liwu Li and external reader Dr. Myron Hlynka for their comments, questions and criticisms which helped me to improve the thesis quality. Special thanks also go to Dr. Xiao Jun Chen for chairing my thesis committee.

I thank my parents who always stand behind me, love me and support me. Thank you, mom and dad.

Special thanks to my beloved wife for her endless love and everlasting support. Without her understanding and contribution to the family, I could not finish the work in time.

Finally, I would like to thank all my friends for their support, advice and help during my thesis work.

Table of Contents

List of Figures

1. Introduction

With the phenomenal growth of the web, there is an ever-increasing volume of information being published in numerous web sites. This vast amount of accessible information has opened up a host of new opportunities so that web mining has become a popular term in the data mining area.

Web mining is the use of data mining techniques to automatically discover and extract information from world wide web documents and services (e.g., on-line travel agents, job listings, electronic malls, etc.). The main difference between data and web mining is the time frame in which analysis happens. In data mining, analysis can take weeks. Web mining deals with data delivered via thousands of mouse clicks per hour, and the analysis takes no more than a few minutes, or perhaps only seconds.

Web mining is more than just searching for resources on the internet, web mining is a conglomerate of techniques for discovering interesting patterns from the large dynamic collection of interconnected resources that form the world wide web. Three distinct research areas constitute web mining discipline: web usage mining for discovering patterns and behaviors from web access logs, web content mining for discovering implicit knowledge from within web documents, and web structure mining, which exploits the presence of links to and from documents to discover pertinent knowledge [MBN+99], [BL99A], [SCD+00].

A general architecture for web usage mining is presented in [MJHS96] and [CMS97]. The architecture divides web usage mining process into two main parts. The first part includes the domain dependent processes of transforming the web data into suitable transaction form. This includes preprocessing, transaction identification, and data integration components. The second part includes the largely domain independent application of generic data mining and pattern matching techniques (such as the discovery of association rules and sequential patterns) as part of the system's data mining engine.

The techniques and corresponding algorithms for web usage mining are of three kinds - associations (in which URLs tend to be requested together), sequential analysis (the order in which URLs tend to be accessed), and clustering (finding natural groupings of users, pages etc.). As in most real-world problems, the clusters and associations in web mining do not have clear boundaries, and often overlap considerably. In addition, bad exemplars (chosen data are outliers or noise) and incomplete data can easily occur in the data set, due to a wide variety of reasons inherent in web browsing and logging. Thus, web mining and personalization requires modeling of an unknown number of overlapping sets in the presence of significant noise and outliers. Moreover, the data sets in web mining are extremely large. In this thesis, we mainly focus on sequential pattern mining techniques, since users' surf patterns are sequential and within a certain time frame.

1.1 Data Mining

Data mining: is the process of analyzing data from different perspectives and summarizing them into useful information. It is the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns from data. Data mining tools can be used to predict future trends and behaviors, allowing businesses to make proactive, knowledge-driven decisions. They scout databases for hidden patterns, and find predictive information that experts may miss because they lie outside experts' expectations.

There are two main kinds of models in data mining: *predictive* and *descriptive*. Predictive models can be used to forecast explicit values, based on patterns determined from known results. For example, from a database of customers who have already responded to a particular offer - like a promotional sale offer, a model can be built that predicts which prospects are most likely to respond to the same offer. Descriptive models describe patterns in existing data, and are generally used to create meaningful subgroups such as demographic clusters. Data mining functionalities include the discovery of concept/class descriptions, association, classification, prediction, clustering, trend analysis, deviation analysis, and similarity analysis [HK00]. A brief description of association rule, sequential pattern, classification and clustering is given in subsequent sections.

1.1.1 Association Rules

Some definitions of Association Rules are given below:

Association rules: Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of binary attributes, called items. Let D be a database of transactions, where each transaction T is a set of items such that $T \subseteq I$. Each transaction T is represented as a binary vector, with $T[k] = 1$ if t bought the item i_k, and $T[k] = 0$ otherwise. Each transaction is associated with an identifier, called TID. Let X be a set of items. A transaction T is said to contain X if and only if $X \subseteq T$. An association rule is an implication of the form $X \Rightarrow Y$ where $X \subset I$, $Y \subset I$ and $X \cap Y = \varnothing$.

Support: The rule $X \Rightarrow Y$ has support s in the transaction set D if $s\%$ of transactions in D contains $X \cup Y$.

Confidence: The rule $X \Rightarrow Y$ holds in the transaction set D with confidence c if $c\%$ of transactions in D that contain X also contain Y.

itemset :a set of items, an itemset that contains k items is called a k-itemset.

Minimum support: the minimum number of transactions in D that an itemset should satisfy to be classified as frequent.

Minimum confidence: the confidence factor c, which is specified by the people, each association rule generated, must have the confidence above this c.

Frequent itemset (Large itemset): all itemsets that have fractional transaction support above minimum support.

Maximal frequent itemset: A frequent itemset is maximal if it is not a proper subset of any other frequent itemset.

A simple example is "60% of customers who purchase beans and rice, also buy oil in the same transaction". This rule is expressed as {beans, rice}→{oil}.

Transaction ID(TID)	*Item*
1	Rice,apple,beans,salt,oil
2	Butter,rice,beans,oil
3	Rice,apple
4	Beans,rice,salt
5	Rice,beans,oil

Figure 1.1.1-1 Customer Purchase Data

Using the simple transaction database in figure 1.1.1-1, the itemset I in this example is {butter, rice, apple, oil, beans, salt}. {rice, beans, oil} occurs in transactions 1,2 and 5. Thus, the support for the set of items {rice, beans, oil}=3/5*100% = 60%. The confidence c can be represented as c% = number of transactions containing X∪Y divided by number of transactions containing X in database. In the example, {rice, beans} occurs in transactions 1, 2, 4 and 5. Thus, the confidence of rule {rice, beans} → {oil} is 3/4*100%= 75%. Itemset {beans, oil, salt} is a set of items {beans}, {oil} and {salt}, and this is a 3-itemset. If the user decides that the minimum support is 3 (60%*5=3) in figure 1.1.1-1, then {rice}, {beans} and {oil} are frequent 1-itemsets. And {rice, beans, oil} is maximal length frequent itemset since they happen together in exactly three transactions.

1.1.2 Sequential Patterns

Association rules aim at discovering co-occurrences at a certain time. With the storage of data over a long time period and development of temporal databases, the discovery of sequential patterns became an important issue. An example of a sequential pattern is "customers typically rent <star wars> then <empire strikes back> and then <return of jedi>. The items in a sequential pattern need not be consecutive but only in that order. The items can be repeated in an access sequence, and every pattern can get support at most once from one access sequence. For example, in the sequence <afbacfc>, fc appears twice, but only one can contribute to the count of fc. Compared with association rule algorithms, sequential algorithms are concerned with finding patterns that contain an ordered set of items rather than patterns that contain an unordered set of items. The algorithms AprioriAll and AprioriSome were first presented in [AS95] to

extract sequential patterns in a transaction database. This work was extended to handle hierarchical information in [SA96]. Since sequential pattern also employs similar concepts like support, minimum support, frequent sequence, therefore, to some extent, sequential pattern is a kind of association rule.

1.1.3 Classification and Clustering

Classification is an important data mining task which can be described as follows. The given data, which are composed of the training set and the test set, consist of multiple examples each having multiple attributes. Each example is tagged with a special class label. The objective is to analyze the training data and to develop an accurate description or model for each class using the attributes presented in the data, and the test set is used to validate the model. The class descriptions are used to classify future data for which the class labels are unknown. They can also be used to develop a better understanding of each class in the data. For instance, a credit card company may classify the credit-rating of its customers into fair or excellent based on a schema such as: CreditInfo (age: integer, income: string, credit-rating: boolean). The database has information on current customers where each record has a customer's age, income and a flag indicating whether a customer's credit-rating is considered fair or excellent. The data can be divided into two parts, one is a training set for building models based on the characteristic of the existing data, the other is a test set to validate the accuracy of the models. Then, the models are used to predict the credit level of new customers whose ages and income are known. The rule "if age is between 31 and 50 and income is high, then excellent credit" is one of many rules that can be deduced from built models.

Unlike in classification, the class label of each object in clustering is not known; Clustering is a technique for grouping data and finding structures in data. The most common application of clustering methods is to partition a data set into clusters or classes, where similar data are assigned to the same cluster whereas dissimilar data should belong to different clusters. There are two major styles of clustering: *partitioning* (often called k-clustering (*k*-means, *k*-medoids)), in which every object is assigned to exactly one group, *and hierarchical clustering*, in which each group of size greater than one is in turn composed of smaller groups. Owing to the huge amounts of data collected

in datasets, cluster analysis has recently become a highly active topic in data mining and web mining research.

1.2 Web Usage Mining

Some definitions and terms relevant to web usage mining are given below:

Web server log: A file that explicitly records the browsing behavior of site visitors.

Click-stream: A click-stream is a sequential series of page view requests. Again, the data available from the server side does not always provide enough information to reconstruct the full click-stream for a site.

User session: A user session is the click-stream of page views for a single user across the entire Web. Typically, only the portion of each user session that is accessing a specific site can be used for analysis, since access information is not publicly available from the vast majority of web servers.

Cookies: Cookies are tokens generated by the web server for individual client browsers in order to automatically track the site visitors

A typical web log entry is like: *137.207.76.3 - [30/Feb/2002:10:03:24 -0100] "GET /users/c/cezeife/courses/60-140/index.html HTTP/1.0" 200 2781,* which consists of: 1) User's IP address: *137.207.76.3*, 2) User ID: Rarely available unless cookie or other technique is used. If this information is not recorded, a hyphen (-) holds the column in the log. In this case, "-" is shown in the log. 3) Access time: *30/Feb/2002:10:03:24 –0100.* 4) Request method: "GET". 5) URL of the page accessed: */users/c/cezeife/courses/60-140/index.html.* 6) Data transmission protocol: *HTTP/1.0.* 7) Return code: *200* (success). 8) Number of bytes transmitted: *2781.*

In contrast to web content and structure mining, utilizing the real or primary data on the web, web usage mining mines the secondary data derived from the interactions of the users while interacting with the web and tries to translate the data generated by web surfer's sessions or behaviors into a certain pattern. The web usage data include data from web server access logs, proxy server logs, browser logs, user profiles, registration data, user sessions or transactions, cookies, user queries, bookmark data, mouse clicks and scrolls, and any other data as the results of interactions [KB00].

The above source data can be obtained from three kinds of collectors:

1) Server level collector, which uses server logs to record the behavior of accessing the specific web sites. In order to trace the cached page views or the information passed through the POST method that can not be recorded in server logs, packet sniffing technology, an alternative method for collecting usage data through server logs, is used to monitor network traffic coming to a web server and to extract usage data directly from TCP/IP packets. The web server also stores other kinds of usage data such as cookies and query data, provides content data, structure information and web page meta-information (i.e. size of a file and its last modification time).

2) Client level collector, which uses a remote agent (such as JavaScript or Java applet) or modifies the source code of an existing browser (i.e. AOL browser) to enhance its data collection capabilities. The benefit of a client side collector over a server side collector is it ameliorates both the caching and session identification problems.

3) Proxy level collector, which acts as an intermediate level of caching between client browsers and web servers, and reduces the loading time of a web page and the network traffic load at the server and client sides [CKR97]. Proxy traces may reveal the actual HTTP requests from multiple clients to multiple web servers. These data sources can be used to characterize the browsing behavior of a group of anonymous users sharing a common proxy server [SCD+00].

There are three main tasks in web usage mining: *preprocessing* like preprocessing of usage, content and structure; *pattern discovery* such as statistical analysis, association rules, clustering, classification, sequential patterns, dependency modeling; *pattern analysis* (used to filter out uninteresting rules or patterns from the set found in the pattern discovery phase) like with SQL and OLAP (online analytical processing).

In general, the above typical data mining methods could be used to mine the usage data after the data have been pre-processed to the desired form [KB00]. However,

modifications of the typical data mining methods such as composite association rules [BL98], an extension of traditional sequence discovery algorithms [BBA+99] and hypertext probabilistic grammars [BL99] are also used.

Applications of Web Usage Mining

The applications of Web Usage Mining could be classified into two categories: learning user profile or user modeling in adaptive interfaces (personalized) and learning user navigation patterns (impersonalized) [KB00], in which five aspects are [SCD+00]:

Web Personalization: Web servers make dynamic recommendations to a web user based on his/her profile and behavior, which can make the web experience of the user personalized to the user's taste, provide the right information and satisfy the need of the user. [MDL+00], [JFM97], [MCS00].

System Improvement: Based on the web traffic behavior, designing different rules for web caching, network transmission, load balancing or data distribution, and detecting intrusion, fraud, attempted break-in [LS98], [MPT99].

Website Modification: changing the structure of a website, clustering pages to determine directly linked web pages [PE97], [SPF99], [MCS99].

Business Intelligence: providing more attractive web pages to different kinds of customers for web shopping, investment, etc. [BM99].

Web Usage characterization: estimating the probability distribution for various pages a user might visit on a given site [MCS00].

1.3 What is Incremental Mining of Sequential Patterns?

When data are inserted or deleted from a database, previous patterns may no longer be interesting and new interesting rules could appear in the updated database. The process of generating new patterns in the updated database (old + new data) by using only the updated part (new data) and previously generated patterns is called incremental mining of sequential patterns.

Although sequential mining is an important data mining task, it has received relatively little attention compared with association rules mining, especially in the area of

incremental mining of sequential patterns [CLK97] [PZO+99] [MPT00]. Existing incremental sequential mining algorithms are apriori-based [AS93] [AS95][PZO+99] [MPT00]. Tree-based mining algorithms [PHM+00] [LE03] are much faster than apriori algorithms and incremental sequential techniques may benefit from a tree-based mining approach.

1.4 Motivation for Thesis

Existing incremental mining of sequential patterns [PZO+99] [MPT00] is based on apriori techniques and tries to reduce the database scan time and the number of generated candidate itemsets. However, it must calculate the candidate itemsets at each level and all of these algorithms have to scan the original database and the updated part of the database a lot of times, which is the most time consuming process.

The WAP-tree structure [PHM+00] is efficient for sequential mining because it only scans the database twice to build a WAP-tree without generating candidate sets, then, it only mines the built WAP-tree which is more compact than the original database and does not scan the original database any more. But when data is inserted into or deleted from the database, the WAP-tree algorithm has to be run from scratch. It is not efficient for frequent maintenance of sequential patterns.

Lu and Ezeife [LE03] proposed the PLWAP algorithm, which is based on WAP-tree algorithm, scans the database twice to build a WAP-tree, then sets up a pre-ordered header node links on the WAP-tree to form a PLWAP-tree. PLWAP eliminates a drawback of the WAP-tree algorithm, namely, recursively re-constructing intermediate WAP-trees involving lots of I/O operations. The PLWAP technique has better performance than the WAP-tree technique making it a better candidate for incremental sequential mining. Again, if we use PLWAP directly for incremental sequential mining, we have to run it from scratch each time the database is updated, which is not efficient.

This thesis proposes two methods for incremental sequential data mining based on the PLWAP-tree that can achieve better performance.

1.5 Contributions of Thesis

In this proposal, two methods named PL4UP (PLWAP FOR UPdated sequential mining) and EPL4UP (Enhanced PLWAP FOR Updated sequential mining) are proposed. These two methods apply the PLWAP-tree to the incremental sequential mining problem. The PL4UP algorithm does not scan the old database even once, the EPL4UP modifies the old PLWAP tree for checking the new added patterns produced from the changed database instead of re-mining whole patterns from a big PLWAP tree built from old DB+ changed db. Both techniques fully utilize old information of original database, like mined patterns, old trees, and do not need to run the algorithm from scratch and thus obtain good performance.

1.6 Outline of Thesis

The rest of the thesis is organized as follows. Chapter 2 reviews existing related work to the thesis. Chapter 3 proposes a detailed description of the new algorithms PL4UP and EPL4UP. Chapter 4 presents implementation and testing while Chapter 5 gives conclusions and discusses future work.

2. Previous/Related Work

2.1 The Sequential Pattern Mining Algorithm Review

2.1.1 AprioriAll and AprioriSome

The algorithms AprioriAll and AprioriSome were first presented in [AS95] to extract sequential patterns in a transaction database. This work was extended to handle hierarchical information in [SA96].

AprioriAll [AS95] follows the same algorithmic steps as the basic apriori algorithm but is for mining sequential patterns. Thus, given a sequential db (e.g. (a) in Figure 2.1.1-1) with a 1-itemset candidate set C_1 ((b) in Figure 2.1.1-1), the AprioriAll algorithm would first compute the frequent 1-itemset L_1 ((c) in Figure 2.1.1-1) with minimum support=2, then, it will generate the C_2 ((d) in Figure 2.1.1-1) as L_1 Apriorigen L_1, prune the sequences that do not meet the apriori property from C_2 before computing L_2 from C_2. Next, it computes C_3 ((f) in Figure 2.1.1-1) from L_2 ((e) in Figure 2.1.1-1), and the process continues until either a Li or a Ci is empty.

Using the same Figure 2.1.1-1 as example, AprioriSome includes two passes: the forward pass and the backward pass. In forward pass it would compute L1 from C1, then, it computes C2 from L1. Now, without computing L2, it computes C3 as C2 apriorigen C2 before it computes L3 from C3. Now that L3 is {B C E}, it applies the apriori property to declare the 2-item sequences {B C}, {B E}, {C E} frequent. Now in the backward phase, it goes to delete all 2-itemsets from C2 that are subset of the set {B C E}, counts the rest of 2-itemsets and prunes with minimum support 2 to get {A C} as the maximal large 2-sequence.

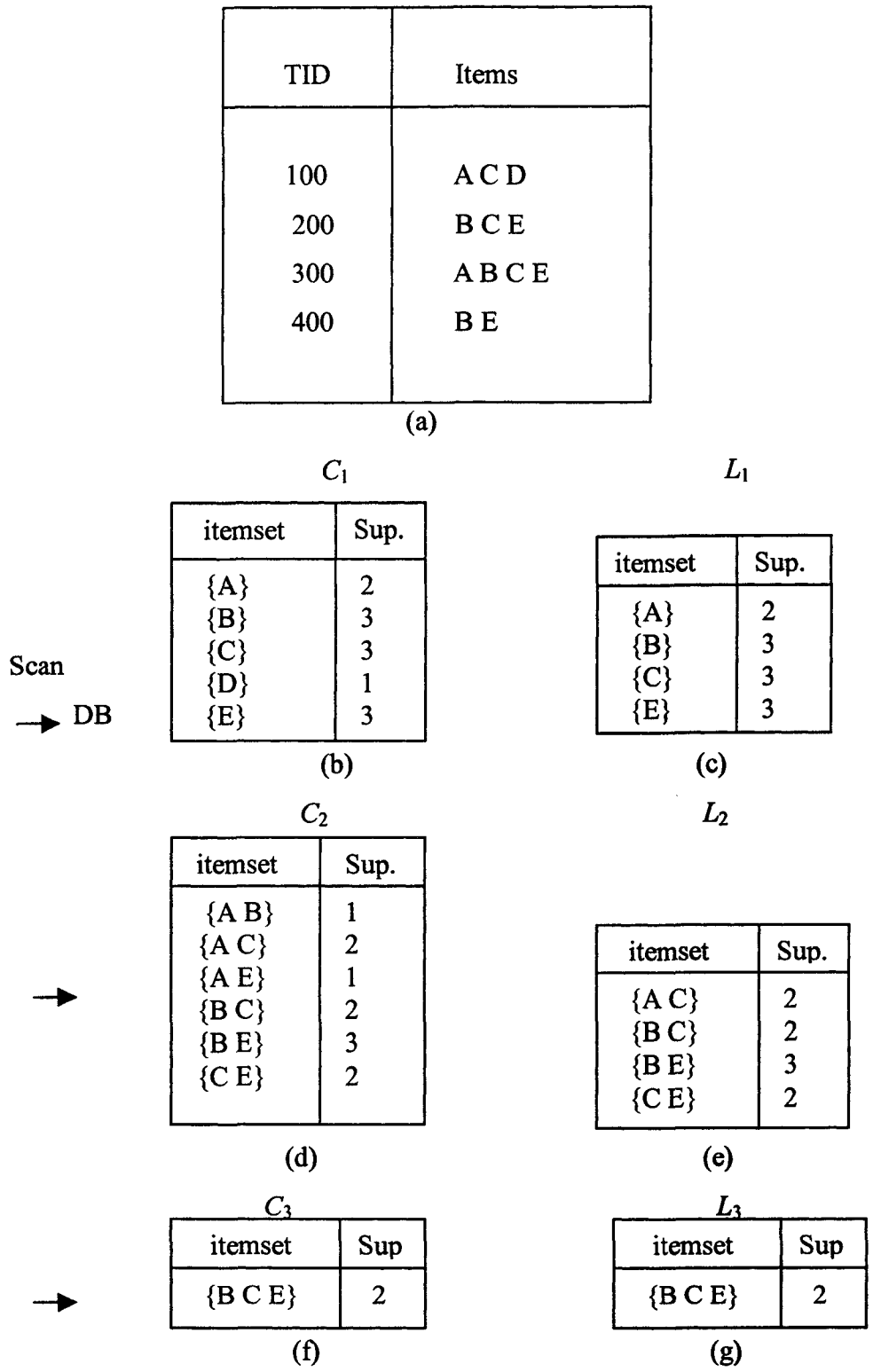


Figure 2.1.1-1 Transaction Database D and Large Itemset Generation

2.1.2 GSP

R. Agrawal and R. Srikant [SA96] proposed GSP (Generalized Sequential Patterns) algorithm to incorporate time constraints, sliding time windows, and taxonomies in sequential patterns, which includes two key procedures: candidate generation (similar to Apriori-gen of Apriori algorithm although details are quite different) and counting candidates (adapting the hash-tree data structure of [AS94]).

GSP makes multiple passes over the database. The first pass determines the frequent 1-itemset L1. This then becomes the seed set for generating the candidate 2-itemset C2. Then GSP passes the database again to count supports of each sequence in C2, prunes C2 with the minimum support to get the large 2-itemset L2 as the seed for generating the candidate 3-itemset C3, repeats the process again until no more candidate itemsets Ci can be generated or there are no frequent sequences at the end of a pass. In the join phase, a sequence s1 joins with s2 only if the last elements of s1 are the same as the first elements of s2. For example, <1 2 3> and <2 3 4> can form the candidate sequence <1 2 3 4> but <1 2 3> and <1 2 4> cannot be joined. In the prune phase, all candidate sequences which have subsequences that do not satisfy the apriori heuristic or have supports lower than the minimum support are removed. e.g. candidate 4-sequence <1 2 3 5> is generated by joining <1 2 3> and <2 3 5> in the large 3-itemset L3. If either of the two subsequences <1 2 5> <1 3 5> of <1 2 3 5> does not exist in the large 3-itemset L3, then <1 2 3 5> should be removed from the candidate 4-itemset C4 before passing over database. Thus, GSP always counts fewer candidates than AprioriAll and is much faster than AprioriAll .

2.1.3 FreeSpan

The Apriori-like sequential pattern mining methods, like their association rule mining methods, suffer from the same problems: they have to generate a huge set of candidate sequences in a large sequence database, scan DB many times, and have more difficulty when mining long sequential patterns. Therefore, [HPM+00] presented a new algorithm, FreeSpan (Frequent Pattern-projected Sequential pattern Mining) to attack

these problems. This runs considerably faster than the Apriori-based GSP algorithm in their experimental performance. The following is the FreeSpan algorithm.

Algorithm1 (FreeSpan) Given a sequence database *S* (the first two columns of Figure 2.1.3-1) and support threshold ξ (min_support=2) , FreeSpan mines the complete set of sequential patterns as follows.

1. Scan *S* once, find the set of frequent items (large-1 itemset L1) in *S*. These frequent items are sorted in support descending order, i.e. <b:5, c:4, a:3, d:3, e:3, f:3>. The list is called the frequent item list or f_list.

2. Construct a frequent item matrix *F* (Figure 2.1.3-2) to count the occurrence frequency of each length-2 sequence formed by items in the f_list, *F* matrix is obtained from a graph on X-Y plane, where the labels on the X-axis represent the f_list, and the labels on the Y-axis is a reverse f_list. *F* is a triangular matrix *F* [j, k], j and k are items in the f_list, *F* [j, k] represents counters of three kinds of combination between j and k like <jk>, <kj>, <(kj)>. e.g. *F* [b, c]=(4,3,0) means that after scanning *S* second time, the supports of <bc>, <cb>, <(bc)> are 4,3,0 respectively. If j=k, *F* [j, k] will only have one counter like *F* [b, b]=4.

Sequence-id	Sequence	Frequent 1-items in each transaction
10	<(bd) cb (ac)>	{a, b, c, d}
20	<(bf) (ce) b (fg)>	{b, c, e, f, g}
30	<(ah) (bf) abf>	{a, b, f, h}
40	<(be) (ce) d>	{b, c, d, e}
50	<a (bd) bcb (ade)>	{a, b, c,d, e}

Figure 2.1.3-1 A Sequence Database

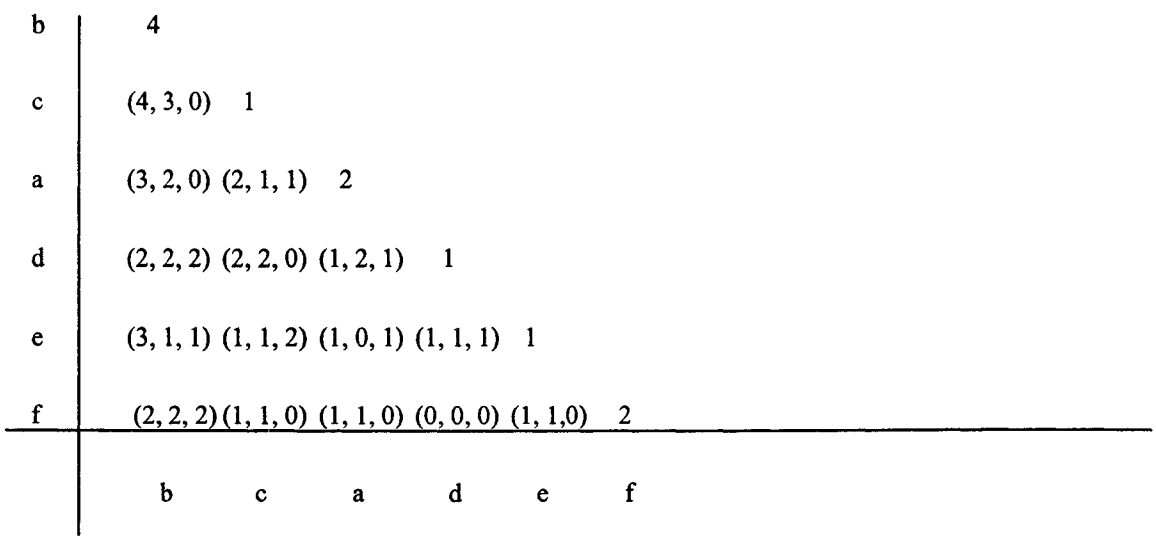


Figure 2.1.3-2 The Frequent Item Matrix after The Scan of *S*

3. Use the matrix to generate: a) length-2 sequential patterns, b) annotations of item-repeating patterns, and c) annotations of projected databases (Figure 2.1.3-3). <...> indicates looking for any particular ordered sequence only and {...} indicates looking for any ordered sequence.

a_i^+ means looking for more than one occurrence of a_i, and a_i :$\{b_p \ldots b_q\}$ represents a set of frequent items which may occur together with a_i to form longer sequential patterns in subsequent mining.

In the matrix table 2.1.3-2, with min_support = 2, the f-row has only *F*[b, f] = (2,2,2) meeting the condition, three length-2 sequences: <bf>:2, <fb> :2, <(bf)>:2, since both *F* [b, b]=4 and *F*[f, f]=2 are frequent, the annotation $\{b^+ f^+\}$ is generated, meaning that one needs to examine multiple occurrences of b's and f's and their combinations in the next scan. Since no other item is co-frequent with <bf>, there is no projected database annotation with f. The e-row has two counters frequent, *F* [b,e] = (3,1,1), *F* [c, e]=(1,1,2), which leads to two length-2 sequences: <be> :3 and <(ce)> :2, since *F* [b, b]=4 but *F* [c, c]=1, *F* [e, e]=1 so, the annotation of item-repeating pattern is $<b^+ e>$. After checking *F*[b, c]=(4,3,0), <bc>:4, a pattern generating triple is formed, the annotation for the projected database should be <(ce> : {b} that indicates generating <(ce> –projected database, with {b} as the only additional items included. Repeating the process, the final table is shown in Fig. 2.1.3-3.

Item	Output length-2 sequential patterns	Ann. On repeating items	Ann. On Projected DBs
F	<bf>:2, <fb>:2, <(bf)>:2	$\{ b^+ f^+\}$	0
E	<be>:3, <(ce)>:2	$< b^+ e>$	<(ce)>:{b}
D	<bd>:2, <db>:2, <(bd)>:2, <cd>:2, <dc>:2, <da>:2	$\{ b^+ d\}< da^+>$	<da>:{bc},{cd}:{b}
A	<ba>:3, <ab>:2, <ca>:2, <aa>:2,	$< aa^+>\{ a^+ b^+\}< c a^+>$	<ca>:{b}
C	<bc>:4, <cb>:3	$\{ b^+ c\}$	0
B	<bb>:4	$< bb^+ >$	0

Figure 2.1.3-3: Pattern Generation from The Frequent Item Matrix

4. After generating the annotations the matrix can be discarded. Only the annotations are used in the third (and the last) scan of the database. Based on the annotation generated from the matrix, scan the DB again and generate the item-repeating patterns and projected databases. The set of item-repeating patterns generated is {<bbf>:2, <fbf>:2, <(bf)b> :2, <(bf)f>:2, <(bf)bf>:2, <(bd)b>:2, <bba>:2, <aba>:2, <abb>:2, <bcb>:3, <bbc>:2}. The projected databases are: <(ce)>: {b}, <da>: {b,c}, {cd}:{b} and <ca>:{b}. For a projected DB which has annotation that contains exactly three items, its associated sequential patterns can be obtained by a simple scan of the projected database. Otherwise, we need to construct a frequent item matrix for the projected database with annotation more than three items and recursively mine its sequential patterns until no more candidate patterns can be found. The final result of the example is shown in Figure 2.1.3-4.

Annotation	<(ce)>:{b}	<da>:{bc}	{cd}:{b}	<ca>:{b}
Projected database	<b(ce)b>, <b(ce)>	<(bd)cb(ac)>, <(bd)bcba>	<(bd)cbc>, <bcd>, <(bd)bcbd>	<bcba>, <bbcba>
Sequential patterns	<b(ce)>:2	<(bd)a>:2, <dca>:2, <dba>:2, <(bd)ca>:2, <(bd)ba>:2, <dcba>:2, <(bd)cba>:2	<bcd>:2, <(bd)c>:2, <dcb>:2, <(bd)cb>:2, <(bd)bc>:2	<bca>:2, <cba>:2, <bcba>:2

Figure 2.1.3-4: Four Projected Database and Their Sequential Patterns

2.1.4 WAP-tree

[PHM+00] proposed an algorithm using a WAP-tree which stands for web access pattern tree. It is quite different from the Apriori-like algorithms and the experimental and performance studies show that the WAP-tree algorithm is an order of magnitude faster than the conventional methods like GSP. There are three main steps for doing the mining:

1. Scan web access sequence DB once to find all frequent events. e.g. in Figure 2.1.4-1, 1-candidate itemsets with their count in DB (the first two columns) are <a:4, b:4, c:4, e:1, f:1>, min_support=3, so, the frequent 1-large itemset L1 is <a:4, b:4, c:4>.
2. Scan DB again, extract frequent subsequence (the third column in Figure 2.1.4-1) in each transaction based on 1-large itemset L_1. The non-frequent part will be discarded. Use only the frequent subsequences as input for constructing a WAP-tree.
3. Recursively mine the WAP-tree using a conditional search.

User ID	Web Access Sequence	Frequent subsequence
100	*Abdac*	*abac*
200	*Eaebcac*	*abcac*
300	*Babfaec*	*babac*
400	*Afbacfc*	*abacc*

Figure 2.1.4-1 A Database of Web Access Sequences

Step 2 for constructing WAP-tree and 3 for mining constructed WAP tree are discussed in more detail next.

Constructing WAP-Tree

When constructing a WAP-tree, a virtual root is inserted first. All other sequences will be the children of the root. Firstly, to insert the first sequence *abac,* it creates a new node (*a*:1) (which means the node is labeled as *a,* and the count is 1) as the child of the root. Meanwhile, the linkage with same event label of *a* is created, then creates a new node (*b*:1) as child of a, also the linkage with same event label of *b* is created. When it creates a new node *a* as a child of *b*, the linkage table already includes *a* and has a link point to the node *a* in the tree. So, it creates a node link from the first node *a* points to the current node *a* and also makes a link from *a* in linkage table to the current *a*, and so on to finish inserting node c:1 below a. Secondly, it inserts the sequence *abcac*, it still start from the virtual root. Since the root has a child labeled *a*, *a*'s count is increased by 1, (*a*, 2) now. Similarly, (*b*, 2) is also in the tree. The next event, *c*, does not match the existing node *a*, and new child node *c*:1 is created and inserted. The remaining insertion process can be processed accordingly. The final tree is show in figure 2.1.4-2.

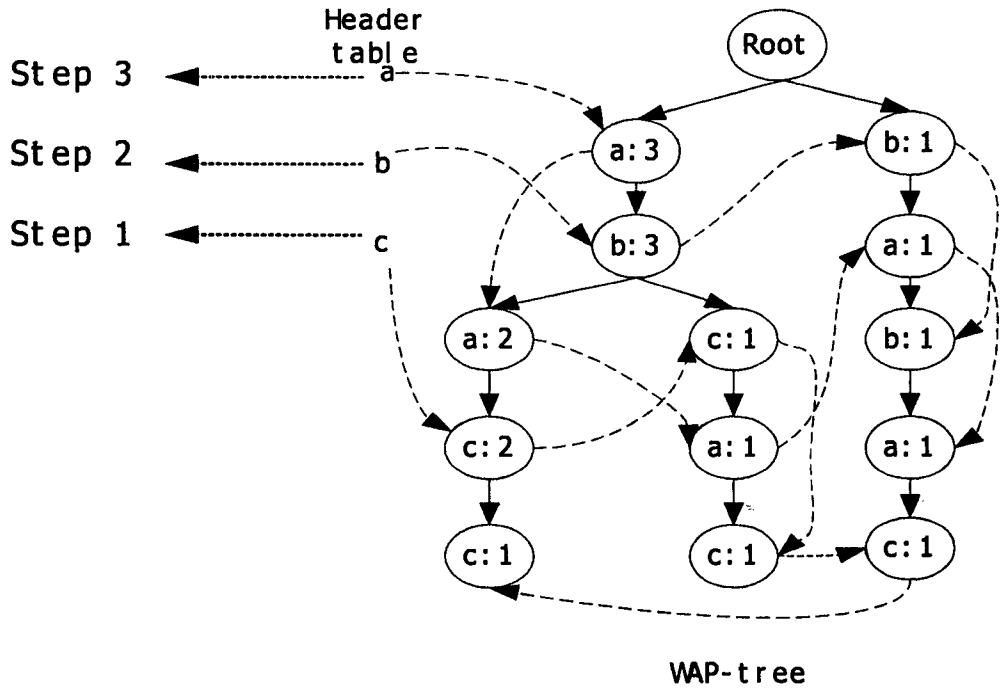


Figure 2.1.4-2 The WAP-tree

Mining WAP-Tree

Using Figure 2.1.4-2 for WAP-tree mining, we first find all sequences which have last event c. It is called the conditional sequence base of *c*:

aba : 2; ab : 1; abca : 1; ab : -1; baba : 1; abac : 1; aba : -1

In the above sequences, there are two sequences ab and aba with count –1. When we find a conditional sequence in a branch of WAP-tree, we need to check whether its prefix subsequence is also conditional sequence of same base. If yes, we need to deduce the count of this subsequence. When we add abca to the conditional sequence of *c*, we find its subsequence ab is also conditional pattern of *c*. Thus, we need to reduce the count of ab by –1. The count we need to deduct from the subsequence is same as the count we add of super-sequence. The list of all events with their counts is a:4, b:4, c:2, min_support=3, so, delete c from the sequence. The rest of the sequences are: *aba:* 2; *ab:* 1; *aba* : 1; *ab* : -1; *baba* : 1; *aba* : 1; *aba* : -1

Using the remaining sequences above and the same tree-constructing procedure, a conditional WAP-tree|c ((a) in Figure 2.1.4-3) is built and recursively obtains the frequent sequences. In this intermediate tree, prefix sequences based on *b* are: *a*:3; *ba*;1. Next, build WAP-tree|bc, only one branch is left, output is abc ((b) in Figure 2.1.4-3), this sub-mining process ends and the mining process goes back to WAP-tree|c to count prefix sequences based on *a*, they are *ab*: 3; *b*: 1; *bab*:1; *b*:-1, then, build WAP-tree|ac ((c) in Figure 2.1.4-3) and recursively obtain the frequent sequence. Using the same procedure, prefix sequences based on *b* are: *a*:3; *ba*:1, build WAP-tree|bac ((d) in Figure 2.1.4-3), only one branch is left, output is abac and end this sub-mining process. Back to WAP-tree|ac, prefix sequence based on *a* is b:1, delete it, build WAP-tree|aac, only root is left, output is aac and this ends the sub-mining process. The mining process goes back to WAP-tree|ac, finds nothing left for next sub-mining, then back to *c*, finds the conditional WAP-tree|c has been mined completely, then it goes to build a conditional WAP-tree|b, mines it, and builds a conditional WAP-tree|a, mines it, then finishes the whole mining process. The frequent sequences obtained for suffix "c" are {c, bc, abc, ac, bac, abac, aac}, the final frequent sequence set are {c, bc, abc, ac, bac, abac, aac, b, ab, a, ba, aba, aa}.

The main drawback of WAP-tree mining is that it has to recursively construct intermediate WAP-trees, which needs more I/O operations and more CPU time.

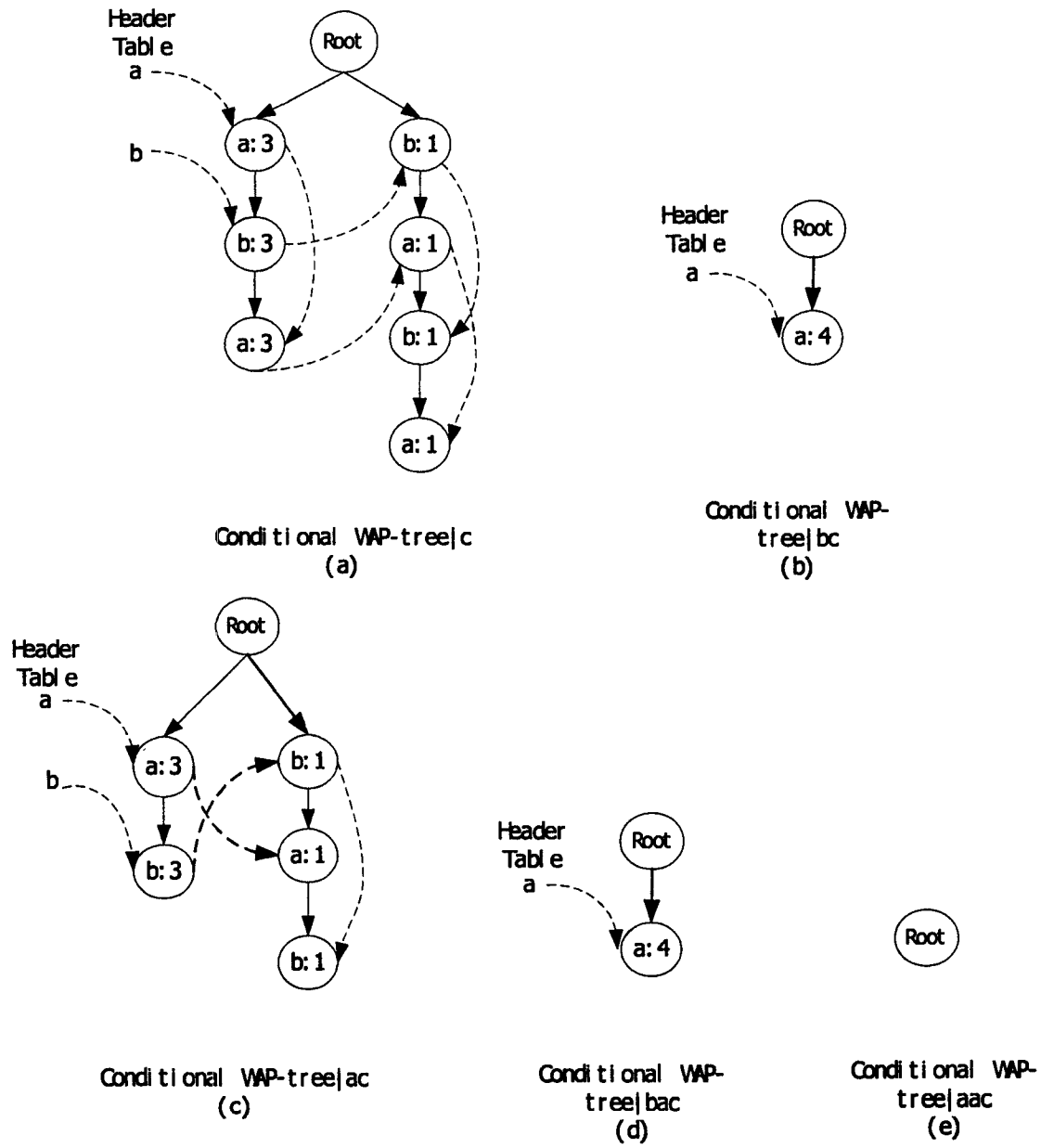


Figure 2.1.4-3 Re-construct WAP Trees for Mining Conditional Base Pattern c

2.1.5 PLWAP

Lu and Ezeife [LE03] proposed a PLWAP algorithm. The basic thinking is to add pre-ordered link on a WAP-tree so that it can avoid reconstructing several intermediate WAP trees during mining as done by the WAP algorithm. We use Figure 2.1.5-1 as example to describe the procedure:

Step1: The PLWAP scans the DB (the first two columns in Figure 2.1.5-1) once to obtain all 1-candidate itemsets (a:4, b:4, c:3, d:1, e:1, f:1). After pruning small itemsets according to min support s(=2), the large-1 itemsets is obtained (a, b, c).

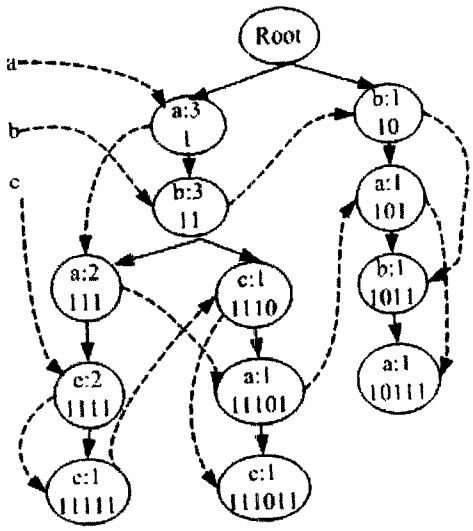
TID	Web access seq.	Frequent subseq.
100	a b d a c	a b a c
200	a e b c a c e	a b c a c
300	b a b a	b a b a
400	a f b a c f c	a b a c c

Figure 2.1.5-1 Original Database DB

Step2: the PLWAP scans the DB a second time to extract the frequent subsequences of each transaction (the third column in Fig.2.1.5-1). Meanwhile, the WAP-tree is built up in Fig. 2.1.5-2. During the WAP-tree constructing, a position code is added to each node, which indicates the position of the nodes in the WAP-tree. The general rule for defining the position code of any node n of the WAP tree is that if the node n is the Root, it has null position code. Otherwise, the position code of node n is obtained by appending '1' to the position code of n's parent node if n is the leftmost child, but appending '0' to the position code of n's nearest left sibling otherwise. For example, to insert the first frequent subsequence abac, start from root, create a new node (*a*:1:1) (which means the node is labeled as *a,* and the count is 1, position code 1) as the left child of root, then create (b:1:11) as leftmost child of *a*. The position code is its parent's position code appending 1. Then create (a:1:111) and (c:1:1111). Then, inserting the second frequent subsequence abcac, since *a, b*, already there only increase their counts. Then, insert *c*. *c* has a sibling *a*, so, its position code is its sibling's position code appending 0, namely 1110. Next insert (a:1:11101) and (c:1:111011). After building the tree, a pre-order traversal mechanism

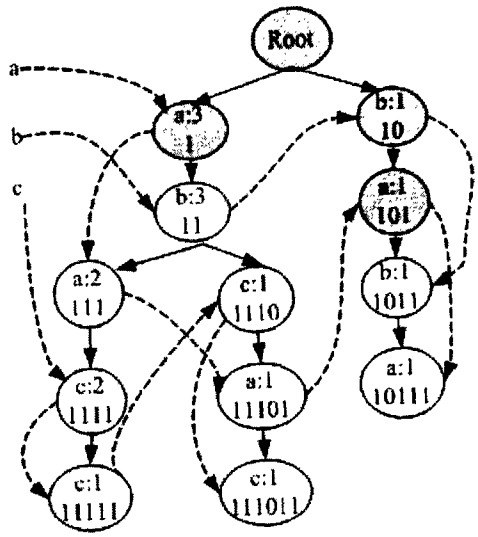
(visit root, visit left subtree, visit right subtree) is used to add a pre-order linkage on the tree. Also, starting from the root, travel the left node first, find a:3:1, which is a root of its suffix tree, so the linkage with same event label of *a* is created. Continue to travel to *a*'s left child b:3:11, and create the linkage too. Then travel to *b*'s left child a:2:111, check the linkage table, find that event *a*'s link was created and points to a:3:1. So, create a link from a:3:1 to a:2:111, also create a link from linkage table to a:2:111. When traveling to c:1:11111 after visiting c:2:1111, find c:1:11111 is at a leaf, so, visit its sibling, find nothing, and go back to its parent c:2:1111, visit c:2:1111's sibling, find nothing, continue back to a:2:111, find *a*'s sibing c:1:1110, create link from c:1:11111 to c:1110, and repeat the previous procedure to finish adding linkage on the WAP-tree.

Step3: the PLWAP begins the mining process. It starts following the header linkage of the first frequent a, and searches the two suffix trees of the PLWAP tree rooted at a:3:1 and b:1:10 respectively to find the first occurrences of 'a' node with total support of 4 from a:3:1 and a:1:101. Because the minimum support is 2, then 'a' is a frequent event to be added to the last list of frequent sequence (∅). Next PLWAP continues to mine all frequent events in the suffix trees of a:3:1 and a:1:101, which are rooted at b:3:11 and b:1:1011 respectively ((b) in Figure 2.1.5-2), PLWAP keeps finding the first occurrences of 'a' for each suffix tree. Then, a:2:111, a:1:11101 and a:1:10111 give 'a' as a frequent event. Thus, *a* is added to the last list of the frequent sequence 'a', to form the new frequent sequence 'aa'. PLWAP continues to mine the conditional PLWAP tree in Figure 2.1.5-2(c). The suffix trees of these *a* nodes which are rooted at c:2:1111, c:1:111011, give another *c* frequent to obtain the sequence 'aac'. The last suffix tree ((d) in Figure 2.1.5-2) is no longer frequent. And this terminates this leg of recursive search. Backtracking in the order of the previous conditional PLWAP tree mined, PLWAP searches for other frequent events. Since no more frequent events are found in the conditional PLWAP tree in figure 2.1.5-2(c), the algorithm backtracks to figure 2.1.5-2(b), b:3:11, b:1:1011and yields a frequent event for *b* to give the next frequent sequence as ab, repeating the same mining procedure. PLWAP gives the mining result as a:4, aa:4, aac:3, ab:4, aba:4, abac:3, abc:3, abcc:2, ac:3, acc:2, b:4, ba:4, bac:3, bc:3, bcc:2, c:3, cc:2.



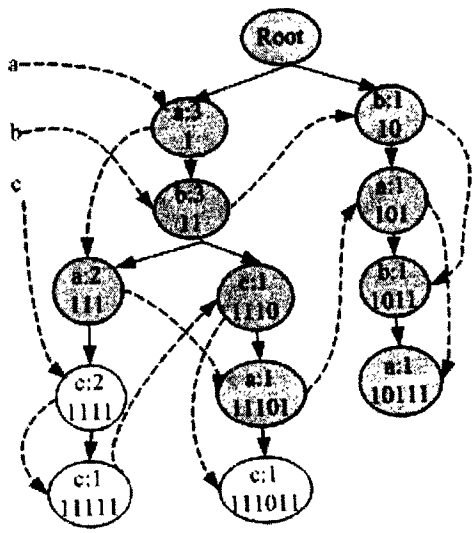
a:{1,111,11101,101,10111}
b:{11,10,1011}
c:{1111,11111,1110,111011}

(a)



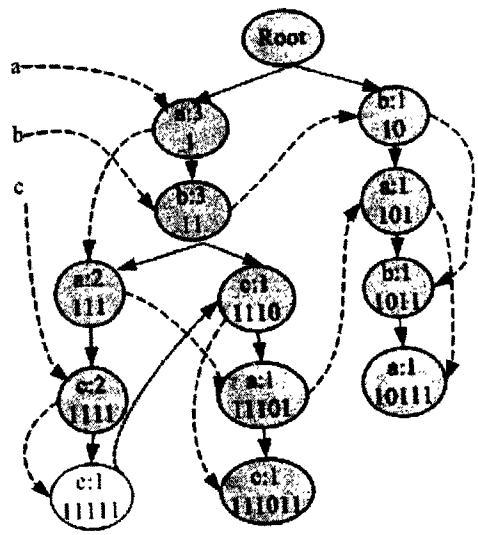
a| suffix tree
{b:11, b:1011}

(b)



aa| suffix tree
{c:1111, c:111011}

(c)



aac| suffix tree
{c:11111}

(d)

Figure 2.1.5-2: Mining PLWAP-tree Starting from *a*

2.2 Incremental Algorithms for Sequential Mining

2.2.1 How the Database Changes during Update

When a database changes during update, it includes two operations: insertion, deletion. Insertion may add new items into existing sequences, e.g. old DB items: a, b, c, d. new additional items: i, j, k. Deletion just deletes old DB items from existing sequences, e.g., old DB items: a, b, c, d. After deleting d, only a, b, c. This also includes adding new sequences or deleting existing sequences. In Figure 2.2.1-1(a), original database: DB (minimum support s = 50%, minimum occurrence =2). The supports of each event in DB are: a:4, b:4, c:3, d:1, e:1, f:1. Original frequent events are a, b, c. After inserting (b) i.e. database db, the updated database (c) is U = DB + db (minimum support s = 50%, minimum occurrence =4). The supports of each event in U are: a:8, b:7, c:3, d:1, e:4, f:5, g:4, h:4, i:1, j:1. Frequent events in U are: a, b, e, f, g, h. Let F and S represent previous large (frequent) items and previous small items in original database respectively; F' and S' represent updated large(frequent) items and updated small items in the updated database U, all events (items) in updated database U will fall into six categories:

1) The items that were large in old DB that are still large in U (F->F'). e.g. a, b in Figure 2.2.1-1.
2) The items that were large in old DB that become small in U (F->S'). e.g. c in Figure 2.2.1-1.
3) The items that were small in old DB that become large in U (S->F'). e.g. e, f in Figure 2.2.1-1.
4) The items that were small in old DB that are still small in U (S->S'). e.g. d in Figure 2.2.2-1.
5) The new items that were not in old DB that become large in U (∅->F'). e.g. g, h in Figure 2.2.1-1.
6) The new items that were not in old DB that are small in U (∅->S'). e.g. i, j in Figure 2.2.1-1

TID	Web access seq.	Frequent subseq.
100	a b d a c	a b a c
200	a e b c a c e	a b c a c
300	b a b a	b a b a
400	a f b a c f c	a b a c c

(a)

TID	Web access seq.	Frequent subseq.
500	a b e g f h	a b e g f h
600	a f b h g i j	a f b h g
700	b a h e f g	b a h e f g
800	a e g f h	a e g f h

(b)

TID	Web access seq.	Frequent subseq.
100	a b d a c	a b a
200	a e b c a c e	a e b a e
300	b a b a	b a b a
400	a f b a c f c	a f b a f
500	a b e g f h	a b e g f h
600	a f b h g i j	a f b h g
700	b a h e f g	b a h e f g
800	a e g f h	a e g f h

(c)

Figure 2.2.1-1: (a)Original Database DB. (b) Inserted Database db and (c)Updated Database U = DB + db

2.2.2 What is the Contribution of an Incremental Algorithm?

Since changes in data become explosive in data source, data mining directly from the original data source becomes more and more difficult. Incremental mining allows data mining using only previously mined frequent patterns and the new changes made to data sources. This will reduce mining response time. Thus, incremental mining becomes one

of the most important methods in data mining. The main contributions of incremental mining algorithm include:

1) Limiting the number of times the old database DB or updated database U are scanned. Since DB or U is usually too big, multiple scanning it is time consuming and not acceptable.

2) Utilizing old information or patterns as much as possible, since some old information or patterns may still be effective in updated database, especially, if the ratio of the changed part is small compared to the old big database.

3) Combining old but still effective patterns with the new additional patterns together to form the whole patterns of the updated database U.

4) Bridging some of the limitations of existing algorithms.

Incremental mining algorithms do not alter the basic thinking of original algorithms. All the existing papers like [CKL97] [PZO+99] [MPT00] [HPY00] [ZE01] [ES02] etc. prove this point. The contribution of all existing papers focuses on above four aspects. While [CKL97] [PZO+99] [MPT00] [ZE01] employ apriori-like algorithm, [ES02] [HPY00] employ tree-like algorithm. Work in [CKL97] [PZO+99] [MPT00] [ZE01] do not change the basic method of apriori, as they scan DB to generate the candidate-1 itemsets, then prune to obtain large-1 itemsets, generate the candidate-2 itemsets from large-1, scan DB to obtain support, then prune to get large-2 itemsets, and so on until no more large itemsets can be generated. However, all of the algorithms can reduce number of scans for huge DB or U, and reduce the number of candidates that need to be checked in each length-k candidate itermsets, thereby reducing total running time through these efforts.

2.2.3 Existing Methods for Incremental Mining of Sequential Patterns

Few works [AS93][AS95] [PZO+99][MPT00] exist on incremental mining of sequential patterns although many researchers work on the algorithms for incremental association rules mining. All known algorithms for incremental sequential patterns

mining are apriori-based rather than tree-based. Two of apriori-based algorithms are described next.

ISM Algorithm (Incremental Sequence Mining)

The ISM algorithm [PZO+99] is based on the sequence lattice (in Figure 2.2.3-3 and Figure 2.2.3-4). The algorithm systematically searches the sequence lattice spanned by subsequent relation. From single items to the maximal sequences in a depth-first manner, the support of each member is kept in the lattice. Also, the algorithm separates the sequence into two sets: the Frequent Set (FS), which denotes the set of all frequent sequences in the updated database, and Negative Border (NB), which is the collection of all sequences that are not frequent but whose subsequences are frequent.

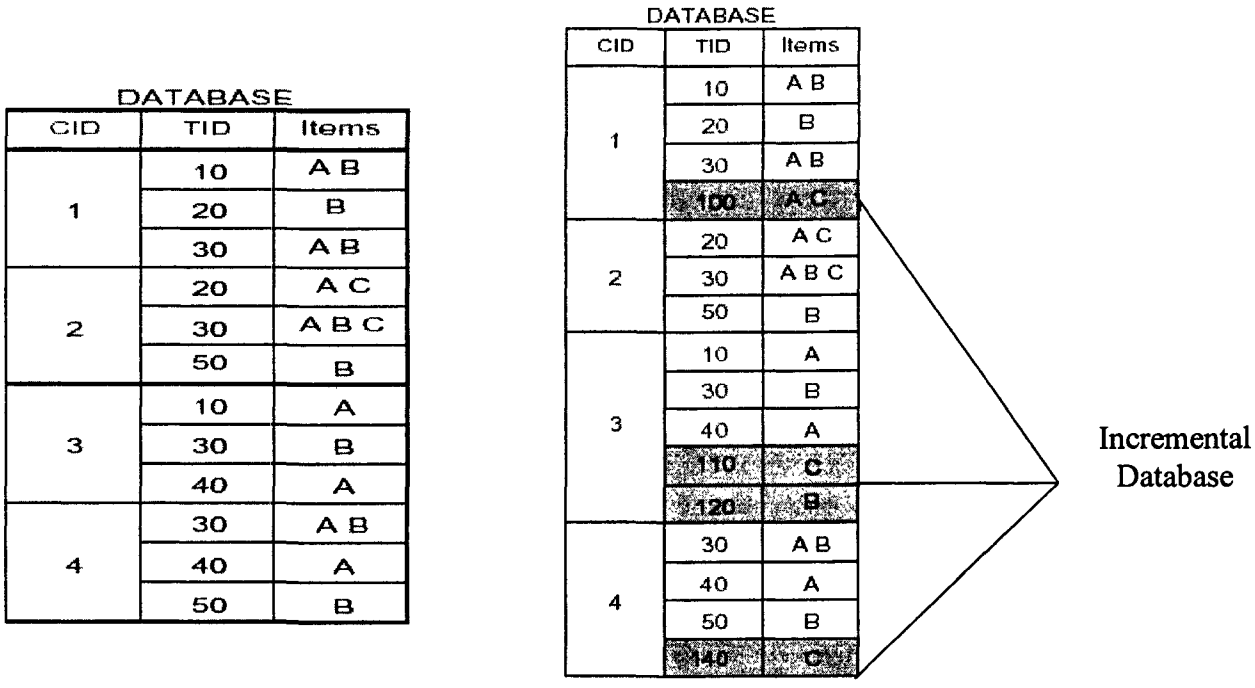


Figure 2.2.3-1 Original and Incremental Database for ISM Algorithm

We briefly introduce the ISM algorithm based on the database (D) and incremented database (D+δ) shown in Figure 2.2.3-1. Let C', T', and I' be the set of all cid's, tid's and items that appear in the incremental part δ. Define D'((a) in Figure 2.2.3-2) to be the set of all records in (D+δ) with cid in C' and D" = D'\δ((c) in Figure 2.2.3-2).

There are 2 phases involved in the ISM.

Phase 1 is for updating the supports of elements in NB and FS.

Support $_{D+\delta}$ (X) = support $_{D}$(X) + support $_{D'}$(X) – support $_{D''}$ (X)

In Figure 2.2.3-1 and Figure 2.2.3-2, Support $_{D+\delta}$ (C) = 1 + 3 – 0 = 4, the old support of (C :1)is recorded in the lattice (Figure 2.2.3-3), and incremental support of (C:3) is obtained through scanning D' and D'', respectively. All of the sequences in the lattice are checked through scanning D' and D'' to update their supports. The following elements' supports are updated: A→A→A (0->1, 0 is support of AAA in D, 1 is support of AAA in D+δ), B→A→A (0->1), A→A→B (2->3), B→A→B (1->2), A→B→B (2->3), and C (1->4). And following moved from the NB to FS: A→A→B, A→B→B, and C.

D'

CID	TID	Items
1	10	AB
	20	B
	30	AB
	100	AC
3	10	A
	30	B
	40	A
	110	C
	120	B
4	30	AB
	40	A
	50	B
	140	C

(a)

δ

CID	TID	Items
1	100	AC
3	110	C
	120	B
4	140	C

(b)

D''

CID	TID	Items
1	10	AB
	20	B
	30	AB
3	10	A
	30	B
	40	A
4	30	AB
	40	A
	50	B

(c)

Figure 2.2.3-2 Databases Derived from Original Database

Phase 2 is for adding to NB and FS beyond what was done in Phase 1. For all 1-sequences that have moved from NB to FS, like C, we intersect it with all other possible

1-sequences. Since C is previous small, the 2-sequences in the lattice do not include C. So, the updated database (D+ δ) has been scanned to get support of A→C(3), B→C(3), C→A(1), C→B(2), AC(2), BC(1) and C→C(1). A→C(3), B→C(3) are added to a table for further generating 3-sequences, the rest of evaluated 2-sequences involving C that are not frequent, which are C→A, C→B, AC, BC and C→C, are placed in NB.

Recursively generating the next k+1 sequences and enumerating the frequent itemsets results in the sequences A->A->C and A->B->C being added to FS $_{D+\delta}$. The sequence AB->C, B->A->C, B->B->C, A->A->A->C and A->A->B->C are added to NB $_{D+\delta}$.

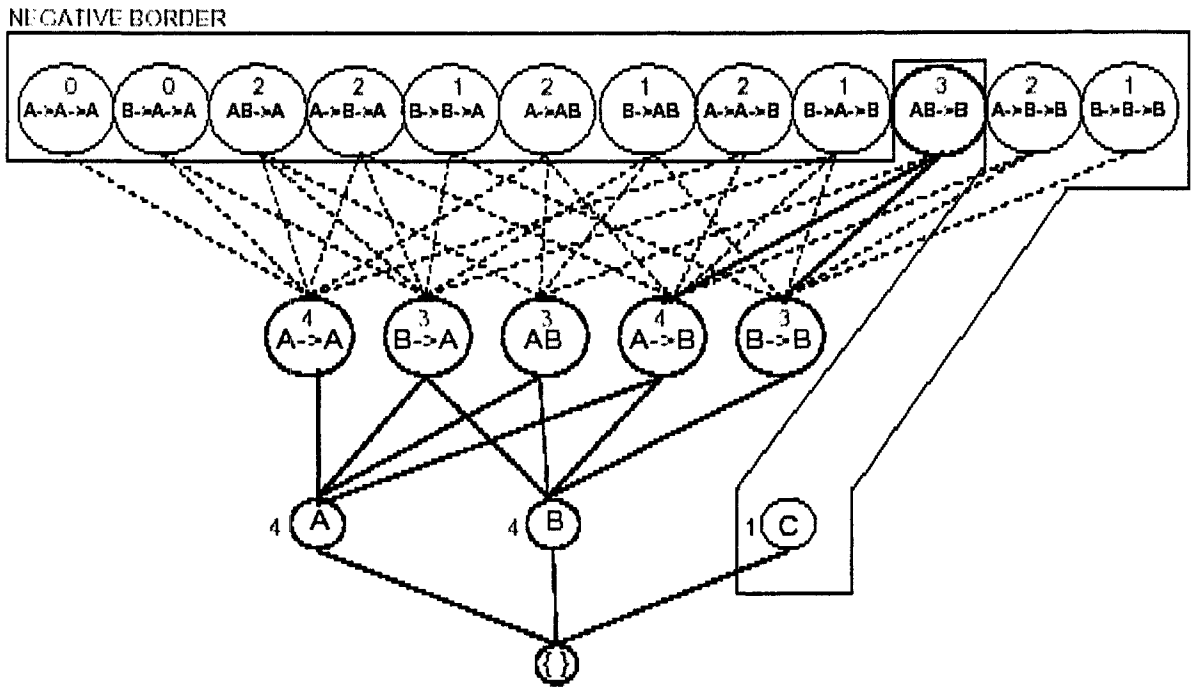


Figure 2.2.3-3 Lattice for the Original Database

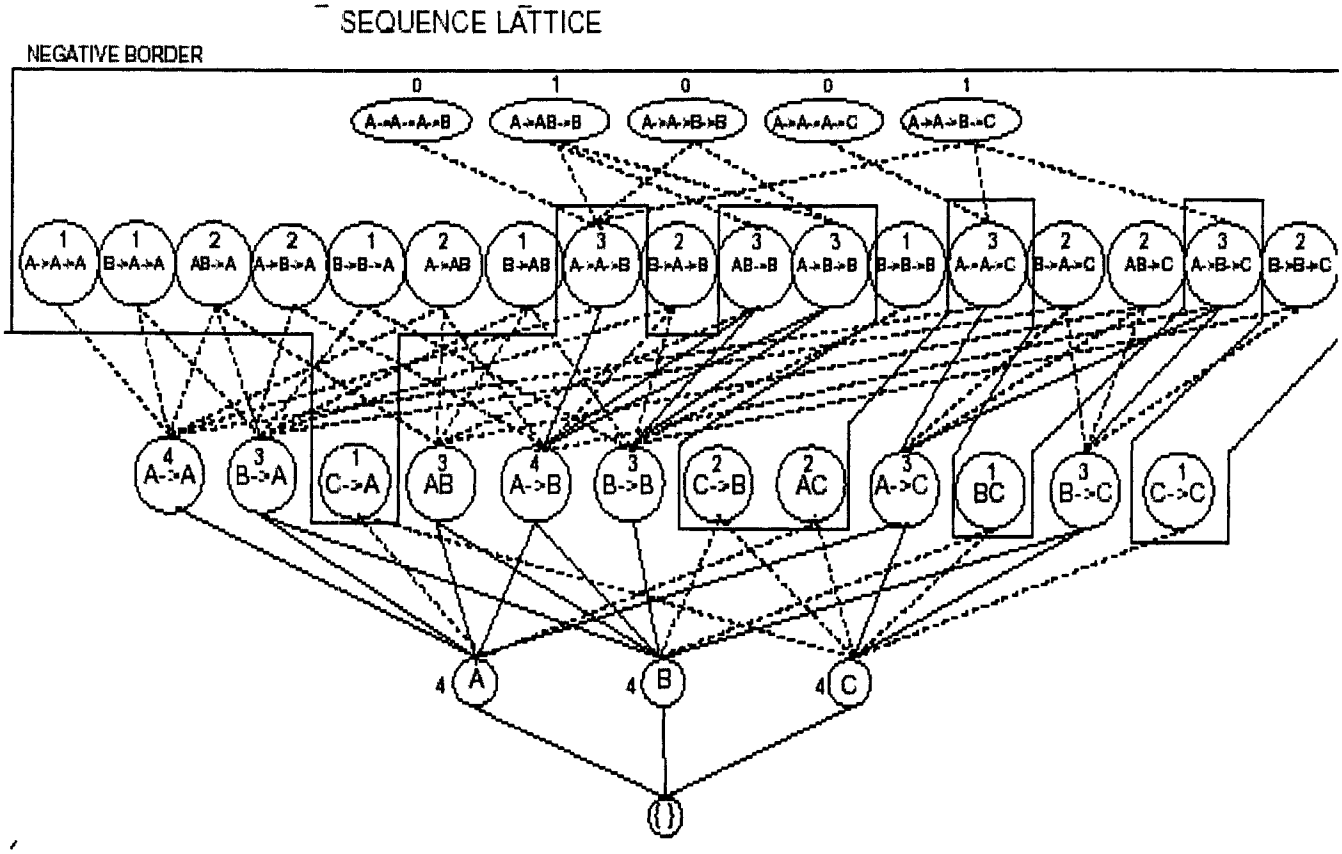


Figure 2.2.3-4 Lattice for Incremental Database

The limitation of the algorithm: it still needs to rescan the updated database D+δ many times if previous small items become large after updating. When the 1-sequence becomes larger, the tree becomes explosive. Although the authors proposed a lattice to separate the tree based on the sequence suffix, the Negative Border can not fit in memory.

ISE Algorithm (Incremental Sequence Extraction)

The ISE algorithm [MPT00b] was proposed by F. Masseglia et al. We discuss it with the following example in Figure 2.2.3-4 (with minimum support =2):

The main process can be decomposed to two steps:

1. Find all *new* frequent sequences of size $j \leq (k+1)$, where k stands for the length of the longest frequent sequences in the original Database. For this step, there are three kinds of frequent sequences considered.

Cust-Id	Itemsets		
C1	10 20	20	50 70
C2	10 20	30	40
C3	10 20	40	30
C4	60	90	

(DB)

itemsets	
50 60 70	80 100
50 60	80 90

(db)

Figure 2.2.3-5 An Original Database (DB) and An Increment Part (db), minsup=2

a. Previous large sequences in the original database that are still frequent in the incremental (inserted) database.
b. New frequent sequences embedded in incremental database but not appearing in the original database.
c. Previous small sequences of original database become frequent when adding items of the incremental database.

First of all, the algorithm scans the incremental (inserted) database db, and gets the set of events that occur at least once in db (item: count): 50:2, 60:2, 70:1, 80:2, 90:1, 100:1. Then after checking with the frequent sets embedded in the original database DB (10:3, 20:3, 30:2, 40:2, 50:1, 60:1, 70:1, 90:1), the algorithm gets the set of frequent 1-sequences which are embedded in db but also frequent in U (DB+db), L_1^{db}: 50:3, 60:3, 70:3, 80:2, 90:2. The candidate generation operates by joining L_1^{db} and L_1^{db} and yields the set of candidate 2-sequences <50 60>, <(50) (60)>, <50 70>, <(50)(70)>,..., <(80)(90)>. Since the candidate <(50) (60)> does not appear in db, it is no more considered when scanning U, only those sets that occur in db at least once are considered. After scanning U, the following sets are put in freqExt (set of frequent sequences obtained from db and validated on the whole database): <(50 60)>, <(50)(70)>, <(50)(80)>, <(60)(80)>. The second step is to find freqSeed, which is obtained by appending to each item of L_1^{db}, its associated frequent sub-sequences. For

example, consider item 70. The frequent item sets in the original DB preceding 70 are <(10), (20), (10 20)>. So, the following sub-sequences are inserted into freqSeed: <(10)(70)>, <(20) (70)>, <(10 20) (70)>. In a similar way, insert the rest of the sub-sequences into freqSeed. The third step is generating candidate set from sequences of freqExt and freqSeed. Consider s'=<(50 60)> from 2-freqEXt and s=<(20)(40)(50)> from freqSeed, dropping 50 from s and appending s' to the remaining sequence. The new candidate sequence <(20)(40)(50 60)> is obtained. Recursively, all the frequent sequences with length $j \leq (k+1)$ are found.

2. Find all frequent sequences of size $j > (k+1)$. This step can be applied GSP-like [SA96] approach at the (k+1) step.

The ISE algorithm mainly relies on the GSP algorithm, and scans the database several times.

2.3 How the PLWAP Works on an Updated Database U?

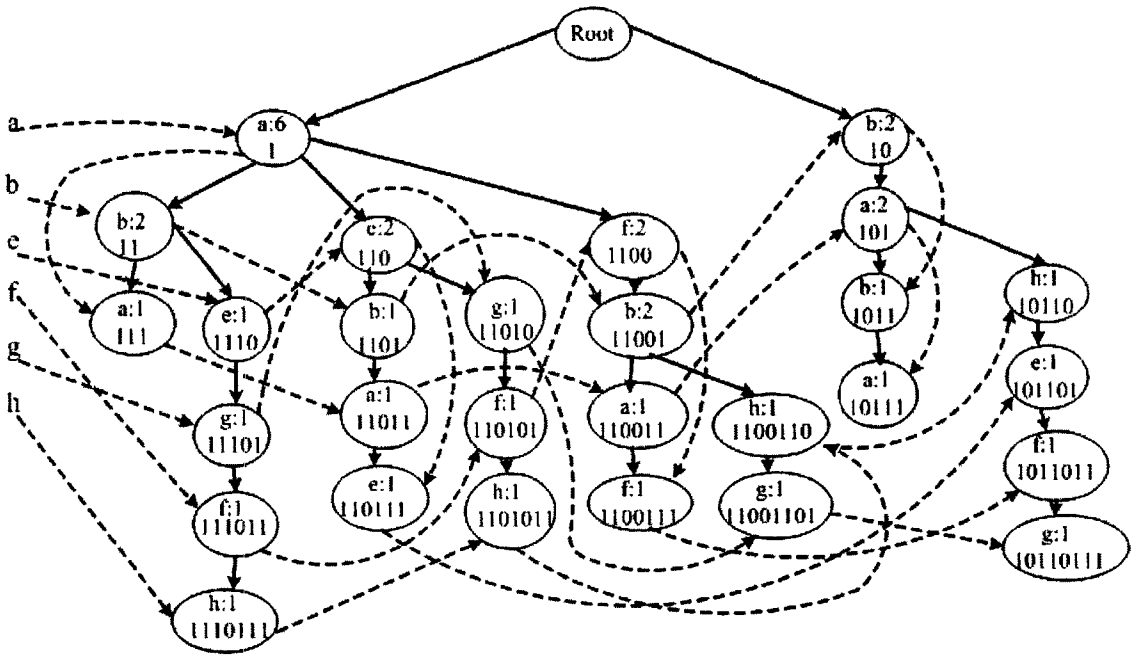
Using the sample in Figure 2.2.1-1, we try to show how the PLWAP works on the updated database U. The min support s = 50%, occurrence is 2 for DB or db, 4 for U. The large-1 itemset are: a, b, c in DB; a, b, e, f, g, h in db; a, b, e, f, g, h in U.

After the database DB is updated, the new changed part db is added into DB and forms a new database U, using the PLWAP algorithm. It has to start from scratch, since in PLWAP, 1) it does not provide any additional method to deal with the old existing patterns, and does not provide any method to treat the U as DB+db or mines DB, db separately. 2) It does not know in which conditions only db can be mined alone without the need to mine the DB, or in which conditions the U could be scanned only once and the mining times of WAP-tree can be reduced. 3) It cannot improve its faults during mining process.

Using the example of Figure 2.2.1-1, the PLWAP mines U in a similar way:

Steps 1 and 2 are the same as the forgoing description. Scan U twice and find all large-1 itemsets, since the c in DB is not large any more, and e, f have become large and new additional large items are g, h. So, in the new built PLWAP-tree, c does not appear any more, and a, b, e, f, g, h form the new PLWAP-tree (Figure 2.3.1-1).

Step 3 is also the same as Figures 2.1.5-2. Starting at link header table *a* and root, check *a* first, then, aa, ab, etc. the final result is: a, aa, ab, aba, ae, af, ag, ah, b, ba, e, f, g, h (Figure 2.3.1-1). The algorithm does not change at all.



a:{1,111,11011,110011,101,10111}
b:{11,1101,11001,10,1011}
e:{1110,110,110111,101101}

f:{111011,110101,1100,1100111,1011011}
g:{11101,11010,11001101,10110111}
h:{1110111,1101011,1100110,10110}

Figure 2.3.1-1 Updated PLWAP-tree (minimum support=4)

2.4 Drawbacks of Existing Incremental Sequential Mining Algorithms and Directly Applying a Tree-based Sequential Mining Algorithm in An Updated Database

The major drawback of existing incremental sequential mining algorithms is that they need multiple scanning of the updated database, which is expensive, and they are all generating large numbers of candidate sequences for pruning, also very time consuming. If we directly apply a tree-based sequential mining algorithm in the updated database, the algorithm must run from scratch, namely, build a big tree (DB+db) from scratch and mine the big tree from scratch, which does not utilize the previous mined patterns, tree, and candidates. Thus, it still wastes some reusable resources. In section 3, we introduce two methods named PL4UP (PLWAP FOR UPdated sequential mining) and EPL4UP (Enhanced PLWAP FOR Updated sequential mining) to deal with the above problem.

3. PL4UP and EPL4UP

After reviewing previous work, we find that using the PLWAP tree method [LE03] to mine incremental database is much faster than the apriori-like method. However, how to mine the incremental part of database and then combine it with the known resources (previous mined patterns ($C1^{DB}$, L1, L2..., PL, RS, SMALL), old tree. etc.) is still a difficult task. We present PL4UP and EPL4UP algorithms which are based on the PLWAP method but have many novel treating techniques to handle the six cases in section 2.2.1 when the database changes. The basic steps are briefly described as follows:

1. Scan the incremental part of database once to get all 1-candidate events (items) C_1^{db} with their counts (occurrences) in db. Combining these events' count in C_1^{db} (1-candidate itemset in changed db) with their corresponding events' count in C_1^{DB} (old database DB), we get C_1' in updated database U (=DB+db). After pruning with minimum support FU (=(DB+db) × s%), we get an updated frequent 1-large itemset L_1.

2. Comparing L_1' with L_1 (in old DB), we can find which case happened when updating the old database DB. Every case except case 3 (S->F') is handled by the PL4UP algorithm. In case 3, PL4UP introduces a tolerance t, which makes part of previous small items to be included in a potential large items group (PL). When events in L_1' are within (L_1+PL), PL4UP still works. When events in L_1' are not included in (L_1+PL), EPL4UP will be introduced to deal with the case.

The following section will use examples to explain how PL4UP works on the six cases (F->F', F->S', S->F', S->S', ∅->F', ∅->S') presented in section 2.2.1 and how EPL4UP works on case 3. All old patterns with their supports, old tree, containing old small items' transactions file (SMALL), are all generated using modified PLWAP method.

3.1 PL4UP

The main idea in PL4UP is:

1. To avoid scanning whole updated database U (DB+db) to build a big PLWAP tree, it scans the changed part, db to build a small PLWAP tree instead. The frequent 1-itemset L_1^{db} is not from pruning 1-candidate C_1^{db} using db × s% but from C_1^{db} intersection with an updated 1-large itemset L_1'. In this way, the non-interesting new patterns cannot be generated from mining the small PLWAP tree. This will save time old patterns compare with new mined patterns to obtain different patterns that need to be checked in the old PLWAP tree.

2. Update the counts of old patterns in the built small tree instead of re-scanning the changed database db to increase their counts as is done by all current incremental mining algorithms. In order to realize the method, a new concept called the Root set list is introduced. A Root set list is used to store the first occurring nodes of each event of 1-large itemset (L_1^{db} in small tree, old L_1 in old tree, updated L_1' in updated old tree) in every branch of a PLWAP tree. When a sequence to be checked comes into the mining process, it first checks whether the first event of the sequence is in a 1-large itemset or not. If not and the sequence's count is less than minimum support, erase it, otherwise, keep it. If yes, search the corresponding root set in the root set list, and use the root set as roots of suffix trees to check the sequence's succeeding events in each branch. If any event's count in the tree plus the sequence's count is less than minimum support, erase it and go to next sequence checking, otherwise, increase the sequence count and keep it in the sequence list.

3. Mine the small tree to get new added patterns (sequences). Compare same length old patterns with new mined patterns, if they are same, erase new mined patterns. If they are not same, put new mined patterns into a remaining sequence list for next step.

4. If the remaining sequence list is not empty, then PL4UP uses the known old PLWAP tree and starts to update the counts of remaining sequences. In order to avoid mining PLWAP tree from scratch, several techniques are introduced:

4.1 Root set list: the same as introduced in step 2, which is used to store the first occurring nodes of each event of old 1-large itemset L_1 in every branch of the old PLWAP tree. The process of checking the remaining sequences is similar to step 2.

4.2 Mask: which is used to avoid checking subsequences that are already included in the validated super sequences. When checking the above remaining sequences, we start checking from the longest sequences k to shorter k-1, k-2, ...2 sequences. After validating the longest sequences k, we start to check k-1 length sequences. Compare k-1 sequences with k sequences first. If any k-1 sequences are subsequences of k sequences, mark them. Then, we check k-1 sequences in the old tree. Any marked sequences will be skipped and the remaining sequences will be checked.

5. After validating the remaining sequences, they will be appended to the old valid sequences as output of final patterns.

The following examples are used to explain PL4UP.

Given the database in Figure 3.1-1, the old itemsets are: old 1-candidates (C_1): a:4, b:4, c:3, d:1, e:1, f:1, the old large-1 items (L_1) with s=50% (minimum support =2): a:4, b:4, c:3. Small-1 items S_1={d:1, e:1, f:1}, the old patterns: a:4, aa:4, aac:3, ab:4, aba:4, abac:3, abc:3, abcc:2, ac:3, acc:2, b:4, ba:4, bac:3, bc:3, bcc:2, c:3, cc:2 and the old tree in Figure 2.1.5-2(a) .

TID	Web access seq.	Frequent subseq.
100	a b d a c	a b a c
200	a e b c a c e	a b c a c
300	b a b a	b a b a
400	a f b a c f c	a b a c c

Figure 3.1-1 Original Database

3.1.1 Case 1: large items that were large in old DB are still large in updated database U(F->F')

1) An inserted database is in Figure 3.1.1-1, scan db once, we get C_1^{db}={a:2, b:2, c:2, d:1, e:1}, combine them with C_1 to form C_1'={a:6, b:6, c:5, d:2, e:2, f:1}, min support FU= 50% *(4+2) =3; large-1 itemsets (L_1') for updated database = {a:6, b:6, c:5}. Small-1 itemsets S_1'= {d:2, e:2, f:1}

TID	Web access seq.	Frequent subseq.
411	a c b d c	a c b c
412	b e c a c	b c a c

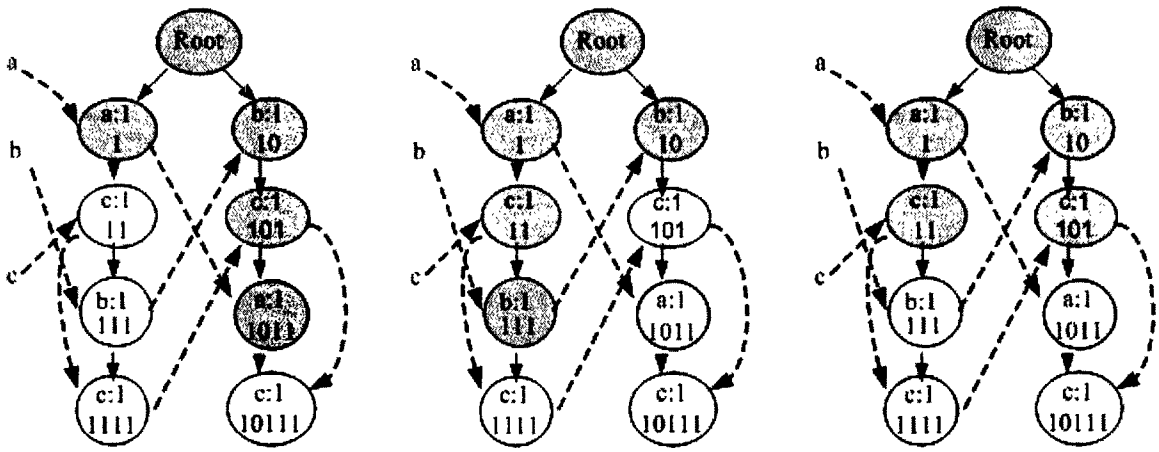
Figure 3.1.1-1: Inserted Database db

2) Find the previous large items that are still large: Compare the L_1 with L_1', the difference between them is equal to ∅ ((L_1'- (L_1'∩L_1)) =∅ and (L_1- (L_1'∩L_1)) =∅), which means previous large items are still large, no previous large items become small, no previous small 1-itemsets become large and no new additional items become large. Then, scan db (new data) to build a small PLWAP tree (Figure 3.1.1-2) using L_1^{db} (a, b, c)= C_1^{db} (a, b, c, d, e)∩ L_1' (a, b, c) instead of using L_1^{db} = db×s%= 2*50%=1(after pruning C_1^{db}, events will be a, b, c, d, e). The linkage table will contain events of L_1^{db} (a, b, c).

3) Update old patterns in the small PLWAP tree: First, find the Root set list for each event in L_1^{db} (also in the linkage table of the small PLWAP tree). Searching events in the linkage table, we find *a* (Figure 3.1.1-2 (a)). Following the link of *a*, we find that a:1:1 is the first occurring node in the branch acbc. Put it into *a*'s Root set list, then follow its link to the next node a:1:1011. Compare the position code of node a:1:1 with node a:1:1011 by using the PLWAP property. a:1:1011 is not child of a:1:1. It also is the first occurring node in the branch bcac. Put a:1:1011 into *a*'s Root set list. Follow a:1:1011's link to find no succeeding node *a* after it. We move to event *b* (Figure 3.1.1-2 (b)) in the linkage table and search all of *b*'s first occurring nodes in all branches of the tree, using the same procedure, *b*'s Root set list includes {b:111, b:10}. Next, *c*'s Root set list includes {c:1:11, c:1:101}. Second, we begin to increase the count of old patterns with sequence length more than one: aa:4, aac:3, ab:4, aba:4, abac:3, abc:3, abcc:2, ac:3, acc:2, ba:4, bac:3, bc:3, bcc:2, cc:2. Input aa:4, check the first event of pattern aa with events in the linkage table. Find event *a* is there. Meanwhile, the corresponding Root set list of *a* {a:1:1, a:1:1011}is also found. Then input the second *a* of pattern aa, search the suffix tree(non-shaded part in Figure 3.1.1-2 (a)) of {a:1:1, a:1:1011}, find nothing below *a*' root set, processing stops. Since aa's count is 4 > minimum support (FU) 3, keep this pattern in OS, and go to check the next patterns aac. In the same procedure, aac:3 is kept.

After checking, all old patterns remain: a:6, aa:4, aac:3, ab:5, aba:4, abac:3, abc:4, ac:5, acc:3, b:6, ba:5, bac:4, bc:4, bcc:3, c:5, cc:4. Only abcc is deleted.

4) Follow PLWAP mining procedure to mine the small tree to get new inserted patterns: a:2, ab:1, abc:1, ac:2, acb:1, acbc:1, acc:1, b:2, ba:1, bac:1, bc:1,bca:1, bcc:1, c:2, ca:1, cac:1, cb:1, cbc:1, cc:2. Compare the new inserted patterns with the old patterns at each same length level but starting from length equal to 2. The new inserted patterns remain: acb:1, acbc:1, bca:1, ca:1, cac:1, cb:1, cbc:1, which will be put in a remaining sequence list for next round check.



Root set list of a:
{a:1, a:1011}
(a)

Root set list of b:
{b:111, b:10}
(b)

Root set list of c:
{c:11, c:101}
(c)

Figure 3.1.1-2 Root set list of L1 ^{db} in Changed Database.

5) Check the above remaining sequence list in the old PLWAP tree to increase their support if applicable. Use the same method described in step 3 (Update old patterns in the small PLWAP tree), find the Root set list (rootSet) for old L_1 (a,b,c) in the old tree (Figure 2.1.5-2(a)). We use the PLWAP position code and linkage, starting from the root, and searching the first occurring nodes of each event of the old 1-large itemset L_1 in every branch of the PLWAP tree to obtain rootSet[a]={a:3:1, a:1:101}, rootSet[b]={b:3:11, b:1:1011}, rootSet[c] = {c:2:1111, c:1:1110 }. Then, we begin to check the longest sequence (4-sequence) acbc: 1. Its first event is *a*, which is in L_1 and the linkage table of the old PLWAP tree, so, use corresponding rootSet[a] as a root set of *a*'s suffix tree to check the succeeding event *c, b, c,* within the suffix tree. Checking *c* in the *a*'s suffix tree, we get {c:2:1111, c:1:111011}, total count of *c* is 2+1=3 (=minimum

support 3). We continue to check *b* in the part below {c:2:1111, c:1:111011} within *a*'s suffix tree to find nothing. So, we need not to check the second *c* of acbc, and the process stops. Since the count of acbc is 1<3, this sequence is deleted from the remaining sequence list. Then, we check the shorter sequence (3-sequence). Since acb, cbc are subsequences of acbc, acbc is not frequent. We still have to check acb, cbc in the old tree because a sequence's sub sequences may be frequent although the sequence is not frequent. Only when a sequence is frequent, can we mask its subsequence to avoid further check. Repeating the same process, we finish checking the sequences in the remaining sequence list to find no sequence in the list is frequent, so the list is empty. The old valid patterns will be final patterns for output. If the list is not empty, we append the additional patterns after the old valid patterns as the final patterns.

3.1.2 Case 2: the items that were large in old DB become small in updated database U (F-> S')

1) Given the original DB in Figure 3.1-1 and an inserted database (the first two columns) in Figure 3.1.2-1, scanning db once, we get C_1^{db}={a:3, b:3, d:1, e:2, f:1, h:1}. Combine this with C_1 to form C_1'={a:7, b:7, c:3, d:2, e:3, f:2 }, minimum support = ceil(50% *(4+3)) =4; large-1 itemsets (L_1') for updated database = { a:7, b:7 }; small-1 itemsets S_1'= {c:3, d:2, e:3, f:2}.

2) Compare L_1 with L_1'. The difference between them is not equal to ∅ ((L_1'-(L_1'∩L_1)) = ∅ and (L_1- (L_1'∩L_1)) = c), which means some previous large items are still large, some previous large items become small, no previous small 1-itemsets become large and no new additional items become large. Then, scan db (inserted database) to build a small PLWAP tree by using L_1^{db} (a, b)= C_1^{db} (a, b, d, e, f, h)∩ L_1'(a, b) instead of using L_1^{db} (a, b, e) pruning from C_1^{db} with minimum support= db×s%=ceil(3*50%)=2.

3) The rest of the procedure is the same as 3) to 5) in the case 1(F->F') in section 3.1.1. But we have to add an extra operation before entering 3) to 5). Since *c* becomes small from large, we have to delete *c* from all old patterns first. Then the remaining old patterns (a:4, aa:4, ab:4, aba:4, b:4, ba:4) are further checked as 3) in the case 1. The final patterns are: a:7, aa:4, ab:6, aba:4, b:7, ba:7.

TID	Web access seq.	Frequent subseq.
421	a f b d a	a b a
422	b e a e b	b a b
423	b h b a e	b b a

Figure 3.1.2-1: Inserted Database db

3.1.3 Case 3: the items that were small in old DB become large in updated database U (S->F')

Given a database DB (the first two columns in Figure 3.1.3-1), the old itemsets are: old 1-candidates (C_1): a:5, b:5, c:3, d:1, e:2, f:2, g:1, h:1. The old large-1 items (L_1) with s=50%(minimum support = ceil(5*50%)=3):a:5, b:5, c:3. Small-1 items S1={d:1, e:2, f:2, g:1, h:1}, the old patterns with s: a:5, aa:4, aac:3, ab:5, aba:4, abac:3, abc:3, ac:3, b:5, ba:4, bac:3, bc:3, c:3 and old tree in Figure 3.1.3-3(a) .

1) If an inserted database is in Figure 3.1.3-2 (the first two columns), scanning db once, we get C_1^{db}={a:2, b:1, e:2, f:2, g:2, h:2}. Combine this with C_1 to form C_1'={a:7, b:6, c:3, d:1, e:4, f:4, g:3, h:3}, minimum support = ceil(50% *(5+2)) =4; large-1 itemsets (L_1') for updated database = {a:7, b:6, e:4, f:4}. Small-1 itemsets S_1'={c:3, d:1, g:3, h:3}.

TID	Web access seq.	Frequent subseq. With s=50%	Frequent subseq. With t=0.8s
100	A b d a c	a b a c	a b a c
200	A e b c a c e	a b c a c	a e b c a c e
300	B a b a	b a b a	b a b a
400	A f b a c f c	a b a c c	a f b a c f c
500	A b e g f h	a b	a b e f

Figure 3.1.3-1 Original Database DB

TID	Web access seq.	Frequent subseq. With s	Frequent subseq. With t
700	B a h e f g	b a h e f g	b a h e f g
800	A e g f h	a e g f h	a e g f h

Figure 3.1.3-2 Changed Database db

2) Compare the L_1 {a, b, c} with L_1' {a, b, e, f}. The difference between them is not equal to ∅ because ((L_1'- (L_1'∩L_1)) = {e, f} and (L_1- (L_1'∩L_1)) = {c}), which means some previous large items are still large, some previous large items become small, some previous small 1-itemsets become large and no new additional items become large. Then, we can not scan db (inserted database) to build a small PLWAP tree like case 1 or 2, mine the small tree, compare new inserted patterns with old patterns, use the remaining patterns to be checked in the old tree because the previous small items e, f (now large) do not appear in the old tree. Thus, related patterns do not exist in the old tree. One way is to scan old database DB once, then scan (DB+db) second time to build a new big PLWAP tree and mine the new big PLWAP tree from scratch, which is directly applying a sequential mining algorithm on an updated database which is not efficient. The second way is to modify the old tree, insert {e, g} into the old tree. This method will be introduced in section 3.2. The third way is to not change the old tree but apply the methods introduced in case 1 and case 2 to case 3. The third way is described below.

Before we build a new PLWAP tree, let us analyze the previous small items. Through observation, the database and its changes, we can find within the relatively small ratio of changed part to original part of the database, that the small items in the old database are divided into two groups. One group has a high possibility of becoming large after an update (the potential large items group PL), and the other has a high possibility of still remaining small (RS) in the updated database. Hence, we introduce a tolerance t, which is smaller than the min support s, t= factor*s. Then, we construct the items with support less than s but more than t into old PLWAP-tree initially. When the database is updated, we just check whether all items that change from small to large are included in the potential large items group PL. If yes, we just build a small PLWAP tree with tolerance t as frequency threshold to mine the changed part to find all possible patterns, and repeat process 3) to 5) in case 1 and case 2 to get final patterns. The tolerance t can be derived from the follow analysis:

Given |DB| as the size of old database, |db| as the size of changed database, minimum support s, the question is: In the worst case, if a previous small item in DB

happens at each transaction in db, how can we choose a tolerance t so that we can include the previous small items in the old patterns and also in the updated patterns?

Solution: if a sequence is frequent in updated database (DB+db), it must satisfy the condition: (|DB|+|db|)*s%. If the support of a previous small item is less than |db|*s%, we simply delete it since it cannot be frequent in (DB+db), only an item with support more than |db|*s% is considered. When the item's support increases in db, the item may meet the condition (|DB|+|db|)*s% even if the item's support in DB is much lower than (|DB|)*s%, in the worst case. If it happens at each transaction in db, then it only needs to happen in t%*|DB| transactions to meet the condition (|DB|+|db|)*s%. So, ((|DB|+|db|)*s% = |db|+|DB|*t%. Then t% = ((|DB|+|db|)*s%-|db|)/|DB|. Let the change rate (CR) of a database can be written as : CR = (|db|/|DB|) *100%. The final t is:

$$t\% = (1+ CR) *s\% - CR. \quad (3.1.3\text{-}1)$$

In general, the formula is: $t\% \leq (1+ CR) *s\% - CR.$ (3.1.3-2)

Also t can be described as: $t \leq F* s$, where F is a constant parameter. (3.1.3-3)

For example, if the change rate CR of database DB is 10%, s% = 50%, then,

t% = (1 + 10%)*50% - 10% = 45 %, so the factor $F \leq t/s = 45/50=0.9$, which means if the change rate of DB is 10%, the range of a tolerance t should be equal or less than 90% of s. If $CR \leq 20\%$, then $t \leq 0.8*s$.

Continuing with Figure 3.1.3-1 and Figure 3.1.3-2, we describe the mining process of PL4UP with tolerance t, s = 50%, minimum support =3, tolerance t = 0.6*s = 30%, minimum occurrences = 2, The old 1-candidates (C_1) are: a:5, b:5, c:3, d:1, e:2, f:2, g:1, h:1. The old large-1 items (L_1) with s=50% are: a:5, b:5, c:3. Small-1 items are S_1={d:1, e:2, f:2, g:1, h:1}. Separate S_1 into PL and RS by using t. Potential large items (PL) in DB={e:2, f:2}, remaining small items (RS) in DB ={d:1, g:1, h:1}. Since the original PLWAP tree includes PL, the old patterns with t (OT) are: a, aa, aac, ab, aba, abac, abc, abcc, abe, abf, ac, acc, ae, af, b, ba, bac, bc, bcc, be, bf, c, cc, e, f. The old patterns with s (OS) are: a, aa, aac, ab, aba, abac, abc, abcc, ac, acc, b, ba, bac, bc, bcc, c, cc.

When the new transactions are added (the first two columns in Figure 3.1.3-2), the algorithm scans db to obtain candidate-1 C_1^{db}={a:2, b:1, e:2, f:2, g:2, h:2}, combines them with C_1 to form C_1'={a:7, b:6, c:3, d:1, e:4, f:4, g:3, h:3 }, min support = 50% *(5+2) = 4; large-1 item (L_1') for updated database = {a:7, b:6, e:4, f:4}. Small-1 items S_1'={c:3, d:1, g:3, h:3}, potential large (PL') ={c:3, g:3, h:3}, remaining small (RS')={ d:1}.

Compare L_1 {a, b, c} with L_1' {a, b, e, f}. The difference between them is not equal to ∅ because ((L_1'- (L_1'∩L_1)) = {e, f} and (L_1- (L_1'∩L_1)) = {c}), but ((L_1'- (L_1'∩L_1)) = {(e, f) ∈ PL) which means new patterns are over the scope of the old patterns with s (OS) but are within the scope of the old patterns with t (OT: a:5, aa:4, aac:3, ab:5, aba:4, abac:3, abc:3, abcc:2, abe:2, abf:2, ac:3, acc:2, ae:2, af:2, b:5, ba:4, bac:3, bc:3, bcc:2, be:2, bf:2, c:3, cc:2, e:2, f:2). Build a small PLWAP tree by using L_1^{db} (a, b, e, f) = C_1^{db} ∩ L_1', delete old patterns containing item c (large to small item), update old patterns with t (OT) in the small PLWAP tree like step 2 in section 3.1, follow PLWAP mining procedure to mine the small tree to get new inserted patterns: a:2, ae:2, aef:2, af:2, b:1, ba:1, bae:1, baef:1, be:1, bf:1, e:2, ef:2, f:2.

Repeat process 3) to 5) in case 1(F->F'). The final patterns are: a:7, aa:4, ab:5, aba:4, ae:4, af:4, b:6, ba:5, e:4, f:4.

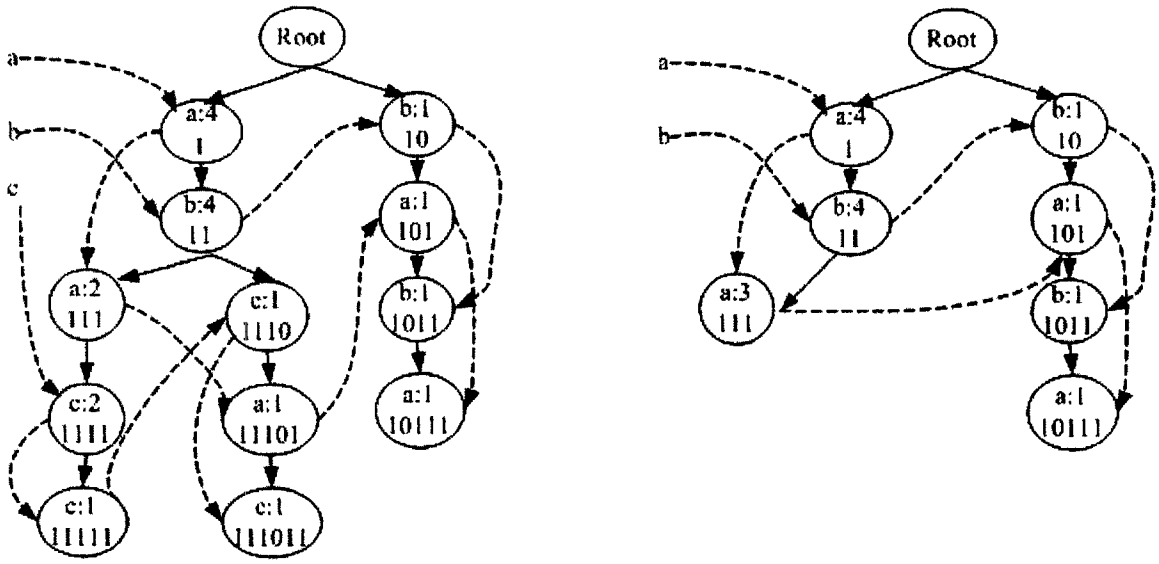


Figure 3.1.3-3 a) PLWAP tree from DB in Figure 3.1.3-1; b) after Delete c

3.1.4 Case 4: the items that were small in old DB and are still small in updated database U (S->S')

Through observation of cases 1(F->F'), 2(F->S'), 3(S->F'), we find that if previous small items in the old DB are still small in updated database U, they do not affect the final patterns. Also this case cannot exist independently. When this case happened, it always caused other cases to happen together, so, PL4UP treats this case the same as other cases with which case 4 is bound.

3.1.5 Case 5: the items that were not in old DB but become large in updated database U (∅->F')

This case also cannot exist independently although it does affect the final patterns. It is always bound with an other case or cases. If it is bound with case 1 or 2, PL4UP treats it the same as case 1 or 2, if it is bound with case 3, but updated 1-itemset L_1' is within the scope of old L_1 and potential large itemset PL, the treating method of PL4UP in case 3 still works. Otherwise, the treating method of EPL4UP in section 3.2 will work on it.

3.1.6 Case 6: the items that are small in db and still small in updated database U (∅->S')

This case also cannot exist independently. It does not affect the final patterns. It is always bound with other cases. It can be handled by using the treating method of PL4UP in corresponding cases.

From analyzing the 6 cases above, we know if a proper tolerance t is chosen, PL4UP can handle all cases by building a separate small PLWAP tree, mining it to get new added patterns, combining with original patterns OT or OS to obtain updated patterns.

If small items are inserted in DB that cause the situation beyond what can be handled by PL4UP, e.g updated 1-large itemset L_1'{a, b, e, f, g, h} is over scope of old L_1{a, b, c} and potential large itemset PL{e}(old small 1-itemset is *d, e, f*, new inserted 1-large itemset is *g, h*) then the algorithm switches to EPL4UP, which is discussed in section 3.2.

3.2 EPL4UP

The main task in EPL4UP is to modify the old WAP tree as follows:

1. Delete previous large items from the tree, change the relationship of the deleted node's parent, son, sibling, and combine branches if necessary.

2. Read the list of previous small items in the old DB, use a 1-itemset from the intersection of the old L_1 and updated L_1' to extract a frequent sequence at each transaction. Then, follow each event of the extracted sequence to search the old tree from root. Reduce the count of the corresponding event in branches of the old tree.

3. Read the list of previous small items in the old DB again, use updated L_1' to extract a frequent sequence at each transaction, follow the tree construction method of PLWAP, and add a new branch for each transaction in the list of previous small items. After constructing the tree, we have to assign a new position code to each node and set up a new linkage table for the tree by using updated 1-itemset L_1' because the old position code and the linkage table are out of date. Then, the tree is ready to be used in next step.

4. The rest of the steps are actually same as the PL4UP steps starting from PL4UP's step 1 to step 5. Build a small PLWAP tree for the changed part db, update old pattern OS in the small tree, mine the small tree to get new inserted patterns, compare these patterns with old patterns. The different patterns will be checked on the modified tree. Root set and mask methods in section 3.1 are also used to reduce the new inserted patterns to be checked. Final patterns are the old valid patterns appending the new remaining patterns.

The following example describes the EPL4UP method.

Given the database DB (the first two columns in Figure 3.1.3-1), the old itemsets are: old 1-candidates (C_1): a:5, b:5, c:3, d:1, e:2, f:2, g:1, h:1. The old large-1 items (L_1) with s=50%(minimum support = ceil (5*50%)=3) are: a:5, b:5, c:3. Small-1 items are S_1={d:1, e:2, f:2, g:1, h:1}. The old patterns are: a:5, aa:4, aac:3, ab:5, aba:4, abac:3, abc:3, ac:3, b:5, ba:4, bac:3, bc:3, c:3 and the old tree is in Figure 3.1.3-3(a), the linkage table list is {a, b, c}.

1) For the inserted database in Figure 3.1.3-2 (the first two columns), scan db once. We get C_1^{db}={a:2, b:1, e:2, f:2, g:2, h:2}. Combine this with C_1 to form C_1'={a:7, b:6, c:3, d:1, e:4, f:4, g:3, h:3}, minimum support = ceil(50% *(5+2)) =4; large-1 itemsets (L_1') for the updated database = {a:7, b:6, e:4, f:4}. Small-1 itemsets S_1'={c:3, d:1, g:3, h:3}.

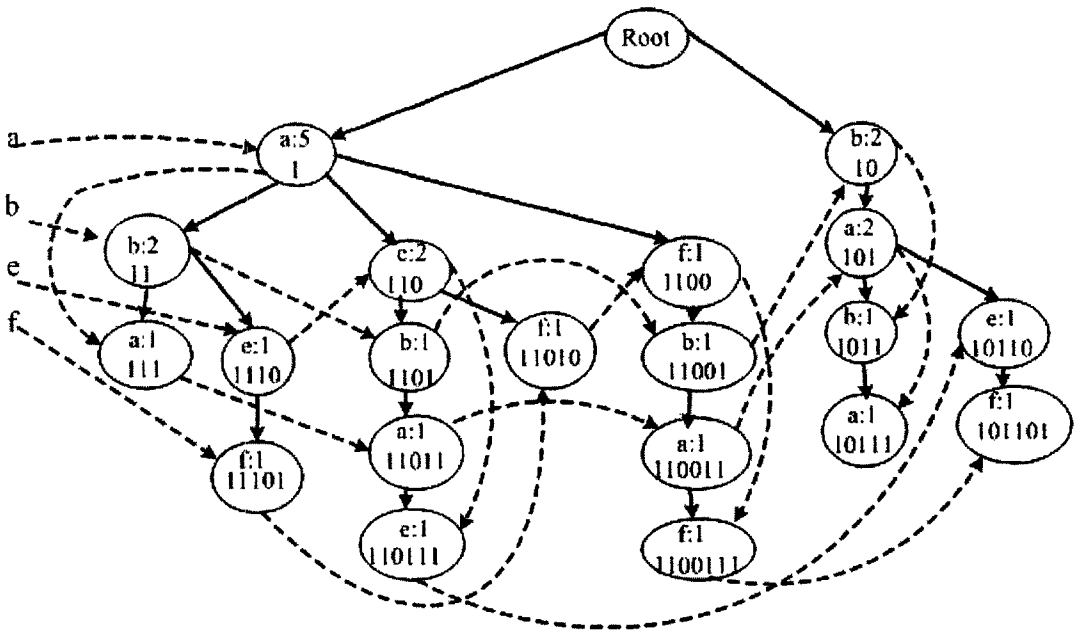
2) Compare L_1{a, b, c} with L_1'{a, b, e, f}. The difference between them is not equal to ∅ because ((L1'- (L1'∩L1)) = {e, f} and (L1- (L1'∩L1)) = {c}), which means some previous large items are still large, some previous large items become small, some previous small 1-itemsets become large and no new additional items become large. Then, because the previous small items e, f (now large) do not appear in the old tree. Thus, the related patterns do not exist in the old tree. EPL4UP is introduced to modify the old tree, insert {e, g} into the old tree.

3) Since {c} is a previous large now small item, it has to be put in a large_to _small's sequence list. Using the sequence list and linkage header table, we try to delete all large_to_small event from the tree. Starting from the first event {c} of the sequence list, search the corresponding event in linkage table {a, b, c}. We find *c* and its link points

to c:2:1111. Since it is not last *c* in the *c*'s link chain, we recursively trace the link chain to the last c:1:111011. Since the last *c* is at a leaf, no child, no sibling, we delete it and make its parent a:1:11101's child pointer point to NULL. Meanwhile, the process backtracks to the last second c:1:1110, it has one child a:1:11101 and sibling a:2:111. They are same event *a*, so, count of a:2:111 is equal to 1+2=3. Delete a:1:11101 since it has no child, and delete c:1:1110. The process backtracks to the last third c: 1: 11111. It is at leaf and has no child and sibling. Delete it and make its parent c:2:1111's child pointer point to NULL. The process backtracks to the first c:2:1111. Delete it since now it is at leaf position and makes its parent a:3:111'child pointer point to NULL. Event *c* has been checked. Then the process checks the next event in the large_to_small sequence list to find nothing and the deletion process ends.

4) Reading the list of previous small items in old DB, there are TID 200 aebcace, TID 400 afbacfc and TID 500 abegfh. Use a 1-itemset called large_to_large sequence list {a, b} derived from intersection of L_1 {a, b, c} with L_1'{a, b, e, f} to extract the frequent sequence of each transaction in the list, TID 200 has aba, TID 400 has aba and TID 500 has ab left. Then, we search the old DB tree from the root to reduce the count of each corresponding event, e.g input *a* of TID 200, find a:4:1, reduce its count to 3, then search *b* below node *a*, find b:4:11, reduce its count to 3, search *a* below *b*, find a:3:111, reduce its count to 2 and so on to finish checking all remaining sequences in the old tree.

5) Read the list of previous small items in old DB again, use updated 1-large itemset L_1' {a, b, e, f} to extract frequent sequence of each transaction in the list and add a new branch on the old tree, TID 200 has aebae, TID 400 has afbaf and TID 500 has abef left. Then, we search the old tree and employ the constructing method of PLWAP to add a new branch for each transaction. After adding the branch, we have to re-assign the position code and linkage table on the tree since the old position code and linkage table already expired during modification. This can be done by travelling the tree once. The final modified tree is shown in Figure 3.2-1.



a:{1,111,11011,110011,101,10111}
b:{11,1101,11001,10,1011}
e:{1110,110,110111,10110}
f:{11101,11010,1100,1100111,101101}

Figure 3.2-1 Modified PLWAP tree after Database Update with minimum s=4

6) The rest of the steps repeat the process 1) to 5) of PL4UP in section 3.1 and final patterns are same as that in section 3.1.3, case 3.

3.3 PL4UP for Transaction Deletion (PL4UP_del)

PL4UP_del is part of PL4UP. The steps of PL4UP_del follow:

1. To avoid scanning whole updated database U (DB-db) to build a big PLWAP tree, it scans the changed part db to build a small PLWAP tree instead. The frequent 1-itemset L_1^{db} is not from pruning 1-cadidate C_1^{db} using db × s% but from C_1^{db} intersection with updated 1-large itemset L_1'.

2. Check whether old patterns with tolerance t (OT) are still valid by using concepts of the Root set list and mask and their processes in section 3.1. t= (1- change rate of DB) × s. The reason to use OT is that it still includes all the patterns in the updated database after the database deletes some transactions and can handle any case in section 3.1 except case 5 and case 6 since they do not exist.

3. After validating the old sequences, they are output as final patterns.

The following examples are used to explain PL4UP_del.

Given database DB (the first two columns in Figure 3.4-1), old itemsets are: old 1-candidates (C_1): a:7, b:6, c:3, d:1, e:4, f:4, g:3, h:3. The old large-1 items (L_1) with s=50%(minimum support = ceil (7*50%)=4): a:7, b:6, e:4, f:4. Small-1 items S_1={c:3, d:1, g:3, h:3}, the old patterns with s(OS): a:7, aa:4, ab:5, aba:4, ae:4, af:4, b:6, ba:5, e:4, f:4. The old patterns with t=0.8*s(OT): a:7, aa:4, aac:3,ab:5, aba:4, abac:3, abc:3, ac:3, ae:4, aef:3, aeg:3, af:4, ag:3, ah:3, b:6, ba:5, bac:3, bc: 3, be:3, bf:3, c:3, e:4, ef:3, eg:3, f:4,g:3, h:3.

TID	Web access seq.	Frequent subseq. With s=50%	Frequent subseq. With t=0.8*s
100	a b d a c	a b a	a b a c
200	a e b c a c e	a e b a e	a e b c a c e
300	b a b a	b a b a	b a b a
400	a f b a c f c	a f b a f	a f b a c f c
500	a b e g f h	a b e f	a b e g f h
700	b a h e f g	b a e f	b a h e f g
800	a e g f h	a e f	a e g f h

Figure 3.3-1 Original database DB

1) If the deleted db that includes TID 700 and TID 800. Scanning db once, we get C_1^{db}={a:2, b:1, e:2, f:2, g:2, h:2}, combine this with C_1 to form C_1'={ a:5, b:5, c:3, d:1, e:2, f:2, g:1, h:1 }, minimum support = ceil(50% *(7-2)) =3; large-1 itemsets (L_1') for updated database { a:5, b:5, c:3}. Small-1 itemsets S_1'= {d:1, e:2, f:2, g:1, h:1}

2) Compare the L_1'{a, b, c} with L_1{a, b, e, f}. The difference between them is not equal to ∅ because ((L_1- (L_1'∩L_1)) = {e, f} and (L_1'- (L_1'∩L_1)) = {c}), which means some previous large items are still large, some previous large items become small, some previous small 1-itemsets become large. But the previous small 1-item {c} is already included in OT, so, rest task is how to check OT in db so that old patterns OT can be updated. We first delete the items that previously large and now are small {e, f} from OT and then scan db (inserted database) to build a small PLWAP tree like case 1 or 2, find the root set for L_1^{db}=L_1'∩C_1^{db}={a, b} in the small tree, input each old patterns OT to check whether it exists in the tree or not. If yes, its count is equal to its count minus the

last event's count, e.g. old ba's count is 5, search the tree, find a branch ba that matches old pattern ba:5, branch ba's last event is *a*, *a*'s count is 1, so, old pattern ba's count now is 5-1=4, in same way, *a*'s count is 7-2=5, *b*'s count is 6-1=5. Then, we prune old patterns OT with updated minimum support=ceil(7-2)*50% =3. The final pattern are a, aa, aac, ab, aba, abac, abc, ac, b, ba, bac, bc, c.

3.4 Algorithm for PL4UP and EPL4UP

The algorithms are divided into four parts: part 1 is main part of PL4UP and EPL4UP (Figure 3.4-1); part 2 is PL4UP algorithm for transaction insertion (Figure 3.4-2); part 3 is EPL4UP algorithm for transaction insertion (Figure 3.4-3); part 4 is PL4UP algorithm for transaction deletion (PL4UP_del) (Figure 3.4-4)

Input: old candidate-1 itemset C_1, large-1 itemset L_1, potential large-1 itemset PL, remaining small-1 RS, minimum support s, potential tolerance t, original patterns with s (OS), original patterns with t (OT), original database DB and its transaction number od, original small item list SMALL, changed database db and its transaction number N. Case number cn.
Output: updated frequent patterns

Begin

1. Read original patterns OS;
2. Scan db first time.
   While (reading each incremental transaction){
   accumulate support (count) of each item (event) i, put them in C_1^{db} ,$count_i = count_i + 1$;
   sequences number N++; }
   for each item i,{
      combine its corresponding count in C_1 to form C_1';
      if $count_i \geq FU\{= (DB+db) \times s\}$, put i into L_1';
      else if $count_i \geq (DB+db) \times t$, put i into PL';
      else put i into RS';}

3. Compare L_1 and L_1':
   put ($L_1' \cap L_1$) into list of large_to_large;
   put ($L_1 - (L_1' \cap L_1)$) into list of large_to_small;
   // case 1 and 2 solution 1
   If $(((L_1' - (L_1' \cap L_1)) = \varnothing)$ switch to PL4UP and pass cn , N;
   //case 3; solution 1
   If $(((L_1' - (L_1' \cap L_1)) \neq \varnothing) \in PL)$
      read original result OT, switch to PL4UP and pass cn , N;
   //case3; solution 2
   If $(\exists ((L_1' - (L_1' \cap L_1)) \neq \varnothing) \in RS))$ switch to EPL4U and pass cn , N;
   //case 5; solution 1
   If $\{((L_1' - (L_1' \cap L_1)) \neq \varnothing) \notin (PL \text{ or } RS))\}$ or $\{((\exists ((L_1' - (L_1' \cap L_1)) \neq \varnothing) \in PL) \&\& \exists ((L_1' - (L_1' \cap L_1)) \neq \varnothing) \notin (PL \text{ or } RS))\}$
      read original result OT, switch to PL4UP and pass cn , N;
   //case 5; solution 2
   If $\{((\exists ((L_1' - (L_1' \cap L_1)) \neq \varnothing) \in RS) \&\& \exists ((L_1' - (L_1' \cap L_1)) \neq \varnothing) \notin (PL \text{ or } RS))\}$ switch to EPL4UP and pass cn , N;

End

Figure 3.4-1 Main Part of PL4UP and EPL4UP Algorithms

Insertion:
Input: case number cn, old large-1 itemset L_1, new large-1 itemset L_1', potential large-1 itemset PL, PL' remaining small-1 itemset RS, RS'(S'=PL'+RS'), minimum support s, tolerance t, original patterns with s (OS), original patterns with t (OT), original database DB and its transaction number od, incremental database db, its transaction number N, 1-candidate C_1^{db}, updated frequency FU, list large_to_large($L_1' \cap L_1$), list large_to_small (L_1- ($L_1' \cap L_1$));
Output: updated frequent patterns

```
Begin
        Restore old WAP tree, put it into root;
        Restore old linkage table, put it into link_header;
        If(large_to_large(L1'∩L1)= ∅){
                cn=1; finalPattern(cn, N)}
        else{
                cn=2; finalPattern(cn, N)}
        if(((L1∪PL)-(L1'∩(L1∪PL))=∅){
                cn=3511; finalPattern(cn, N)}
        else{
                cn=3512; finalPattern(cn, N)}
End

FinalPattern (cn, N){
        If(cn = 2 or 32 or 52)
                Delete events in old sequences according to events in the list of large_to_small;
        1-large itemset L1^db =L1'∩C1^db;
        scan db again to build small PLWAP tree and linkage table using L1^db;
        update counts of old patterns in the small PLWAP tree{
                if( count of old sequences < FU) delete it;
                else  large_to_large=old sequences ∪ large_to_large;}
        min_support F=s×N;
        min_ support with tolerance T=t×N;
        if(cn=1 or 2 or 32 or 52)
                PLWAP mining process(PLWAP tree, linkage table, updated patterns AP, F);
        Else
                PLWAP mining process(PLWAP tree, linkage table, updated patterns AP, T);
        Compare AP and OS||OT and put remaining part of AP into list of remaining_new_seq;
        Mining_Old_Tree(root, link_header, old L1){
                Find root set for L1';
                Compare super sequence with sub sequence of remaining_new_seq to mask sub
                    sequence of the valid super sequence;
                mining remaining_new_seq according to found root  set, put result into AP;}
        AP=AP∪ large_to_large;

Return pattern AP }
```

Figure 3.4-2 PL4UP Algorithm for Transaction Insertion

Insertion:
Input: case number cn, original PLWAP tree, old large-1 itemset L_1, new large-1 itemset L_1', potential large-1 itemset PL, PL' remaining small-1 RS, RS' (S'=PL'+RS'), minimum support s, potential tolerance t, original patterns with s (OS), original patterns with t (OT), original database DB and its transaction number od, the list of previous small items SMALL, inserted database db and its transaction number N, list large_to_large($L_1' \cap L_1$), list large_to_small ($L_1 - (L_1' \cap L_1)$);
Output: updated frequent patterns

```
Begin
        Restore old WAP tree, put it into root;
        Restore old linkage table, put it into link_header;
        Updated_WAP_tree(list SMALL,  list large_to_large, list large_to_small,
                        L1' , root, link_header){
          Sub_update {
            For each event ei  in  large_to_small
             If (ei is not last node of ei in the tree) call Sub_update;
             Else if (ei .son=NULL) delete ei, ei.parent.son =NULL;
               Else If (ei. son = ei .sibling){
                        ei.sibling.count= e .sibling.count+ ei.son.count
                         delete ei..son, and el}
                   else ei.parent.son= ei.son, delete ei ;}}

        While(read list SMALL){
           Extract frequent sub sequence in each transaction if its event el ∈ (L1∩L1');
           For each ei in the sub sequence{
           search the tree, if tree's node.event = el;  node.count =node.count-1}

        While(read list SMALL){
           Extract frequent sub sequence in each transaction if its event el ∈ L1';
           Constructing WAP tree;
           Re-assign position code and linkage table;}

        finalPattern(cn, N);

End
```

Figure 3.4-3 EPL4UP Algorithm for Transaction Insertion

Deletion:
Input: case number cn, old large-1 itemset L_1, new large-1 itemset L_1', potential large-1 itemset PL, PL' remaining small-1 itemset RS, RS'(S'=PL'+RS'), minimum support s, tolerance t, original patterns with s (OS), original patterns with t (OT), original database DB and its transaction number od, incremental database db, its transaction number N, 1-candidate C_1^{db}, updated frequency FU, list large_to_large($L_1' \cap L_1$), list large_to_small (L_1- ($L_1' \cap L_1$));
Output: updated frequent patterns

Begin

Read original pattern OT;
Scan db first time.

While (reading each incremental transaction){

accumulate support (count) of each item (event) i, put them in C_1^{db} ,$count_i$ = $count_i$ +1}

for each item i,{

combine its corresponding count in C_1 to form C_1';

if $count_i \geq FU\{= (DB-db) \times s\}$, put i into L_1';
else if $count_i \geq (DB-db) \times t$, put i into PL';
else put i into RS';}

finalPattern (cn, id);

End

finalPattern (cn,id){
scan db again to build PLWAP tree and linkage table using L_1^{db};
Mining_New_Tree(root, link_header, L_1^{db}){
Find root set for L_1^{db};
While(mining remaining_old_seq according to found root set){
If(find remaining_old_seq in the tree)
(remaining_old_seq.count)--;}
If(count of old sequences < FU) delete it;
else large_to_large= remaining_old_seq $\cup$ large_to_large;
updated patterns AP=AP$\cup$ large_to_large; }

Return pattern AP }

Figure 3.4-4 PL4UP Algorithm for Transaction Deletion (PL4UP_del)

4. Experimental Evaluation and Performance Analysis

In this section, we compare the performance comparison of PL4UP and EPL4UP with existing algorithms ISE and PLWAP. All experiments are performed on a 1.8GHz Pentium 4 PC machine with 256 megabytes memory. The operating system is Windows 2000. All algorithms are written in C++ language and running under Microsoft Visual C++ environment. Section 4.2 and 4.3 give results for database insertion, while section 4.4 gives results for database deletion. The change rate of the database is 10% of the original database for insertion or deletion.

4.1 Dataset

Our synthetic data sets are generated using the publicly available synthetic data generation program of the IBM Quest data mining project at: http://www.almaden.ibm.com/cs/quest/. The data sets consist of sequences of events, where each event represents a web page for web accessing. The parameters shown below are used to generate the data sets.

|D|: Number of sequences in database

|C|: Average length of the sequences

|S|: Average length of a maximal potentially frequent sequence

|N|: number of events

For example, C10.S5.N2000.D60k represents a group of data with average length of the sequences is 10, the average length of a maximal potentially frequent sequence is 5, the events in the database are 2000, and the total number of sequences in the database is 60K. If these four parameters become larger, the execution time becomes longer.

The original results like old patterns with s (OS), old patterns with t (OT), old tree, etc. are generated using the PLWAP method. Since they are not the part of incremental algorithm, the running time is not included when calculating the execution time of PL4UP and EPL4UP.

Since our algorithm automatically determines when to switch to PL4UP or EPL4UP, both of them do not appear together at same time. In order to check the performance of PL4UP and EPL4UP, the inserted database is same size but have different sequences in the inserted database. The tolerance t of PL4UP for insertion or deletion (PL4UP_del) is chosen to be 80% minimum support. That is, if the minimum support is 1, then the tolerance is 0.8.

4.2 Experiment 1: Execution Time for Different Support

This experiment uses a fixed size database and a fixed size of inserted (or deleted) transactions, given different minimum support, to compare the execution time of PL4UP, EPL4UP with ISE and PLWAP. The data sets are described as C10.S5.N2000.D20K+2k. The minimum support is increased from 1% to 5%. The experimental results are shown in Figure 4.2-1, Figure 4.2-2 and Figure 4.2-3.

Algorithms	Runtime (in seconds) at different support (%)							
	1	2	3	4	5	16	18	20
ISE	4004	1405	515	188	101	137	113	100
PLWAP	212	112	80	66	60	32	30	29
PL4UP	164	69	40	27	14	—	—	—
EPL4UP	—	—	—	—	—	23	21	19

Figure 4.2-1 Execution Time for Different Algorithms

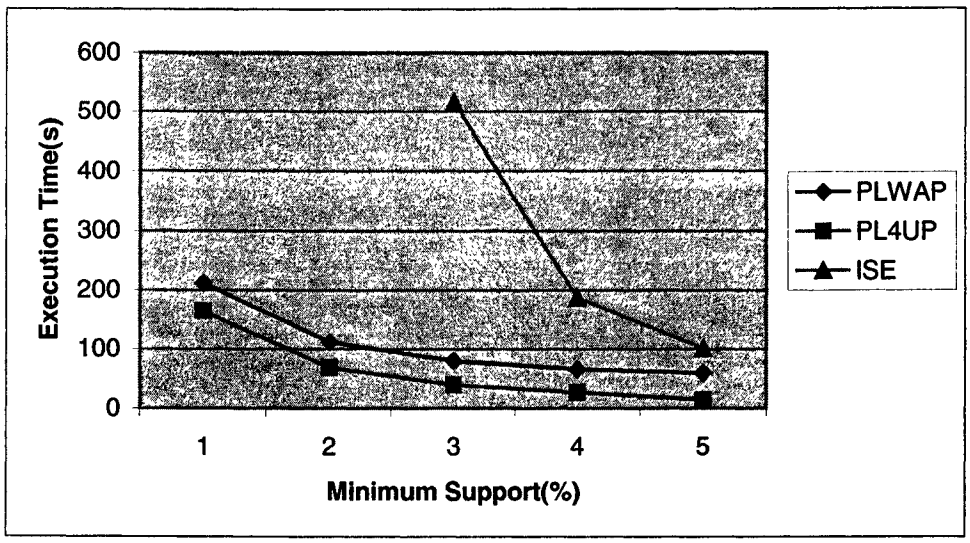


Figure 4.2-2 Execution Time with Different Minimum Support for PLWAP, PL4UP and ISE

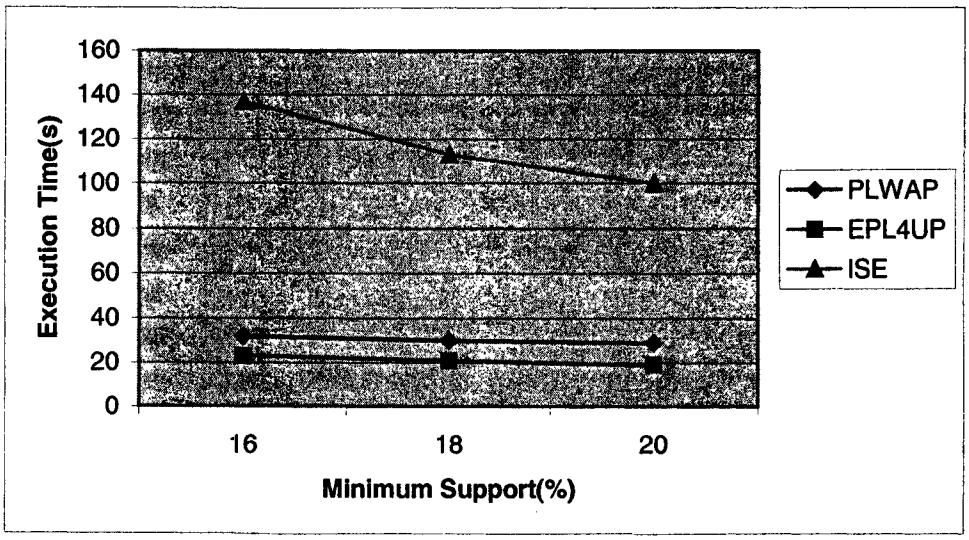


Figure 4.2-3 Execution Time with Different Minimum Support for PLWAP, EPL4UP and ISE

From Figure 4.2-2, and Figure 4.2-3, we can see that all the execution times decrease as the minimum support increases, which means the number of candidate sequences decreases. Thus, the algorithms need less time to find the frequent sequences.

Among these four algorithms, ISE is the most time-consuming algorithm. When the minimum support is lowed, the candidate itemsets to be generated increases

exponentially at each level. If the 1-large itemset L_1=20, 2-candidates itemset generated from L_1 is 400. If half of them need to scan the old DB to count their supports and 50 of them are confirmed large, the next 3-candidates itemset generated from L_2 will be 2500 and so on. The testing is very time-consuming.

The rest of the three algorithms are all based on a pre-order position coded tree structure. They do not need to scan old DB many times to check candidate sequences. So, they are all faster than the apriori-like ISE, but PLWAP has to scan the old DB and inserted db from scratch when the database updates so that it needs more time than PL4UP and EPL4UP.

4.3 Experiment 2: Execution Time for Databases with Different Size

We use different size databases that vary from 20k to 100k to compare the four algorithms. The minimum support 1% and 5% are used for PL4UP, ISE and PLWAP in Figure 4.3-1, Figure 4.3-2, Figure 4.3-3 and Figure 4.3-4. Minimum supports of 16% and 20% are used for EPL4UP, ISE and PLWAP in Figure 4.3-5, Figure 4.3-6, Figure 4.3-7 and Figure 4.3-8. The five datasets are C10.S5.N2000.D20k, C10.S5.N2000.D40k, C10.S5.N2000.D60k, C10.S5.N2000.D80k, and C10.S5.N2000.D100k.

	Runtime (in seconds) at different data size				
Algorithms	20k	40k	60k	80k	100k
ISE	4004	7589	—	—	—
PLWAP	212	384	546	789	1018
PL4UP	164	244	327	410	504

Figure 4.3-1 Execution Data with Different Data Size on Minimum Support 1%

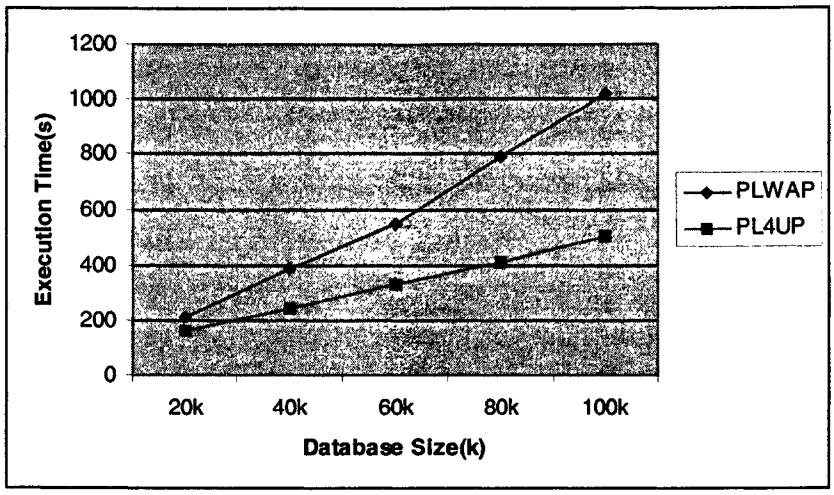


Figure 4.3-2 Execution Time with Different Data Size on Minimum Support 1%

Algorithms	Runtime (in seconds) at different data size				
	20k	40k	60k	80k	100k
ISE	101	202	302	404	505
PLWAP	60	128	187	279	461
PL4UP	14	24	37	65	115

Figure 4.3-3 Execution Data with Different Data Size on Minimum Support 5%

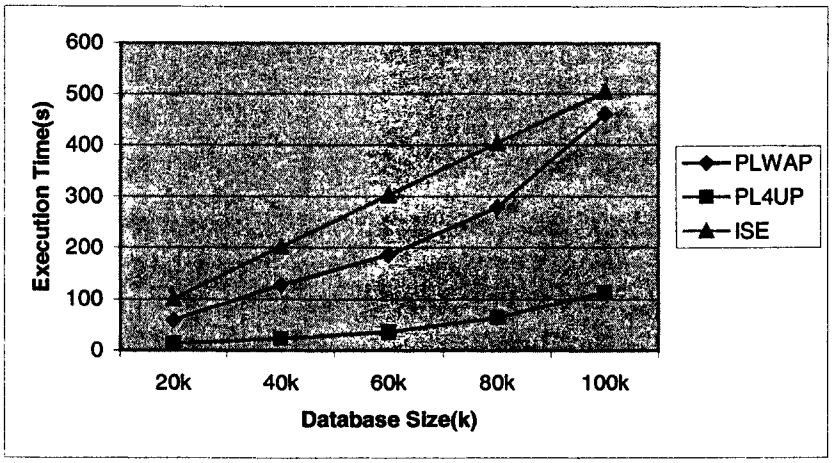


Figure 4.3-4 Execution Time with Different Data Size on Minimum Support 5%

Algorithms	Runtime (in seconds) at different data size				
	20k	40k	60k	80k	100k
ISE	137	265	404	530	653
PLWAP	32	61	91	117	157
EPL4UP	23	40	72	106	136

Figure 4.3-5 Execution Data with Different Data Size on Minimum Support 16%

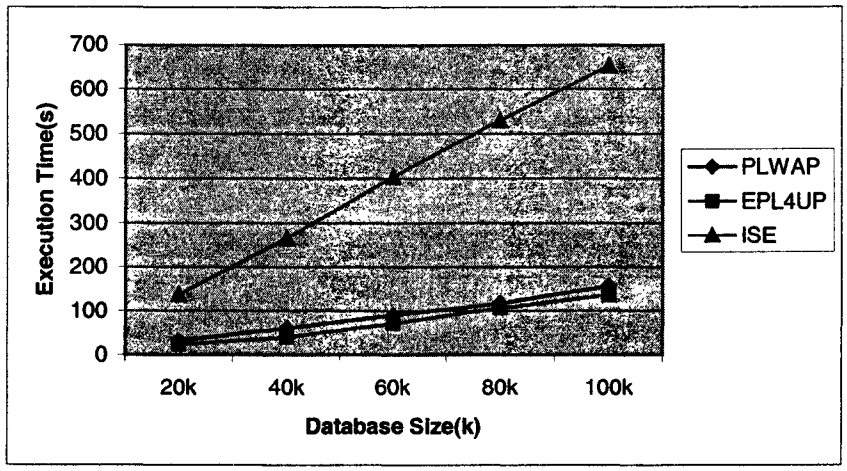


Figure 4.3-6 Execution Time with Different Data Size on Minimum Support 16%

Algorithms	Runtime (in seconds) at different data size				
	20k	40k	60k	80k	100k
ISE	100	189	285	378	470
PLWAP	29	51	81	98	128
EPL4UP	19	33	54	83	105

Figure 4.3-7 Execution Data with Different Data Size on Minimum Support 20%

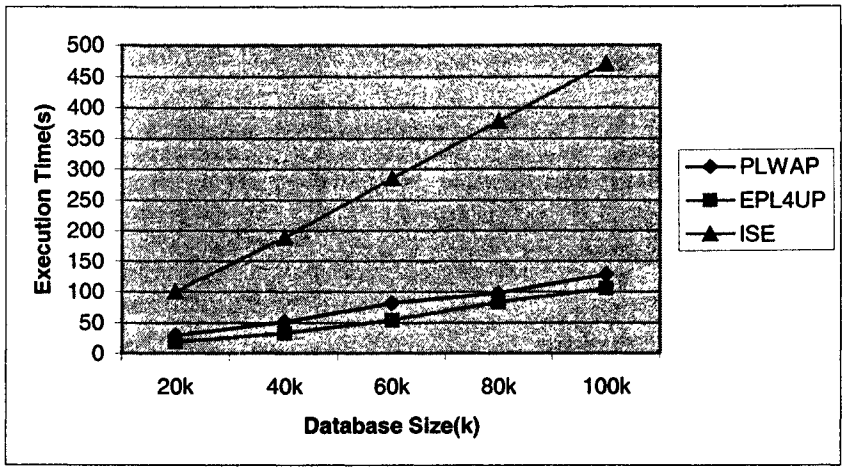


Figure 4.3-8 Execution Time with Different Data Size on Minimum Support 20%

From the result of Figure 4.3-1 to 4.3-8, we observe that:

1) When the inserted transactions become large, all the execution times become larger.

2) The larger amount of changes of database size affect the execution time of ISE and PLWAP with lower minimum support much more than with higher minimum support. When minimum support s is 1%, the difference in execution time between PLWAP and PL4UP is 1018-504=514 seconds for 100k data but the difference in them is 461-115=346 when s=5%, which explains that when we lower minimum support, it causes more previous small items and small subsequences to become large, so, the PLWAP tree size increases and causes longer checking time. When the minimum support increases from 16% to 20%, the difference in execution time between PLWAP and EPL4UP decreases since the PLWAP tree varies to small and checking time reduces.

3) The PL4UP and EPL4UP always obtain better performance than PLWAP and ISE.

4.4 Experiment 3: Execution Time for Database Deletion

In this experiment, we use the same databases as in section 4.2, which vary from 20k to 100k. The minimum support 1% and 5% are used for PL4UP_del and PLWAP in Figure 4.4-1, Figure 4.4-2, Figure 4.4-3 and Figure 4.4-4 respectively. Since ISE did not provide any methods for deletion, we only compare PL4UP_del with PLWAP.

Algorithms	Runtime (in seconds) at different data size				
	20k	40k	60k	80k	100k
PLWAP	192	335	483	627	860
PL4UP_del	97	169	240	309	378

Figure 4.4-1 Execution Data with Different Data Size on Minimum Support 1%

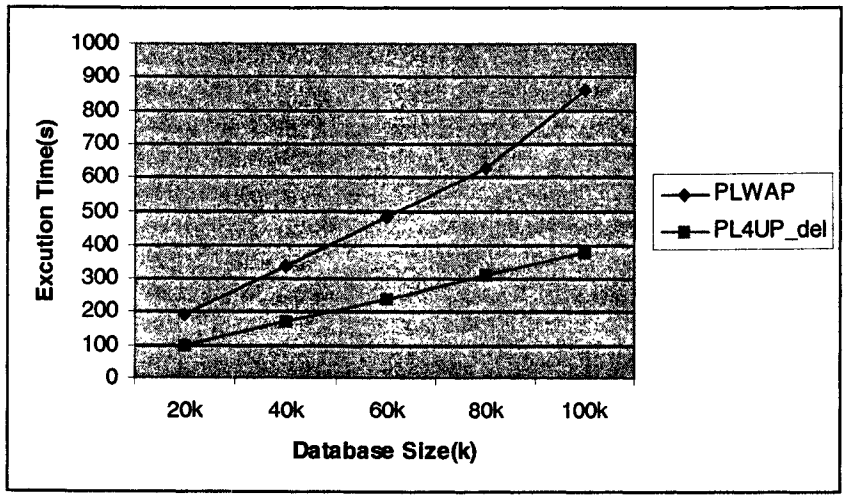


Figure 4.4-2 Execution Time with Different Data Size on Minimum Support 1%

Algorithms	Runtime (in seconds) at different data size				
	20k	40k	60k	80k	100k
PLWAP	55	105	203	274	379
PL4UP_del	12	23	32	43	55

Figure 4.4-3 Execution Data with Different Data Size on Minimum Support 5%

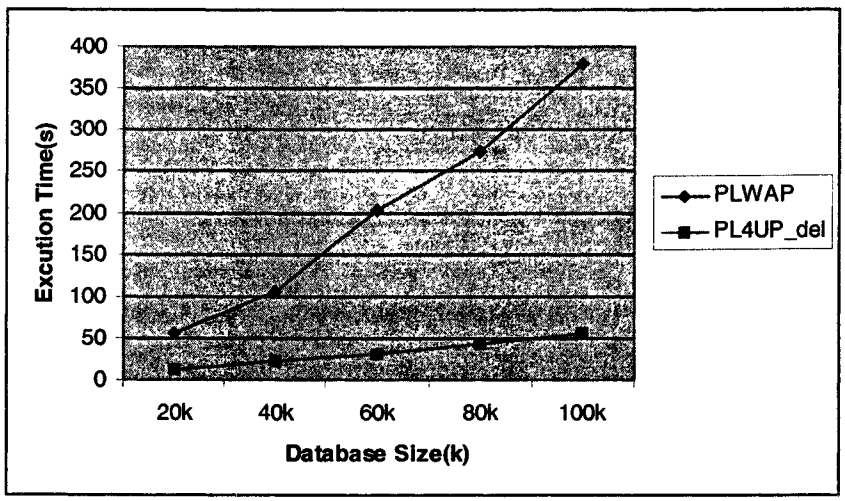


Figure 4.4-4 Execution Time with Different Data Size on Minimum Support 5%

From the result shown in Figure 4.4-2 and 4.4-4, the varied trend of the curves is the same as the insertion of section 4.2 and 4.3. We clearly find PL4UP_del is more efficient than PLWAP. As the size of database increases, the difference of execution time between PLWAP and PL4UP_del gets bigger.

4.5 Correctness of Algorithm Implementations

To show that the implementation of the four algorithms is correct, three small sample databases are used to test the correctness of the implementations. Figure 4.5-1 and Figure 4.5-2 are for PL4UP, ISE and PLWAP. Figure 4.5-1 and Figure 4.5-3 are for EPL4UP, ISE and PLWAP. Figure 4.5-4, and Figure 4.5-2 are for PL4UP_del and PLWAP. Minimum support is 50%, tolerance t is 30%.

TID	Events
100	10 20 40 10 30
200	10 50 20 30 10 30 50
300	20 10 20 10
400	10 60 20 10 30 60 30
500	10 20 50 70 60 80

Figure 4.5-1 Original DB for PL4UP, EPL4UP, ISE and PLWAP

TID	Events
700	20 10 80 50 60 70
800	10 50 70 60 80

Figure 4.5-2 Insert (delete) db for PL4UP (PL4UP_del), ISE and PLWAP

TID	Events
600	5 10 60 20 80 70
700	20 10 80 50 60 70
800	10 50 70 60 80

Figure 4.5-3 Insert db for EPL4UP, ISE and PLWAP

TID	Events
100	10 20 40 10 30
200	10 50 20 30 10 30 50
300	20 10 20 10
400	10 60 20 10 30 60 30
500	10 20 50 70 60 80
700	20 10 80 50 60 70
800	10 50 70 60 80

Figure 4.5-4 Original DB forPL4UP_del and PLWAP

After running three algorithms PL4UP, ISE and PLWAP against the databases in Figure 4.5-1 and Figure 4.5-2, the results of frequent sequences are shown in Figure 4.5-5. The results of these three algorithms are the same. The only difference is the order of sequences produced from the algorithms. Similar results are obtained in running

EPL4UP, ISE and PLWAP in Figure 4.5-1 and Figure 4.5-3, running PL4UP_del and PLWAP in Figure 4.5-4 and Figure 4.5-2. The results are shown in Figure 4.5-6 and Figure 4.5-7 respectively.

Algorithms	Frequent Sequence Results
ISE	{10},{20},{50},{60} {10,50},{10,60},{10,10},{10,20},{20,10} {10,20,10}
PLWAP	{10},{10,10},{10,20},{10,20,10},{10,50},{10,60} {20},{20,10} {50} {60}
PL4UP	{10},{20},{50},{60} {10,10},{10,20},{10,50},{10,60},{20,10} {10,20,10}

Figure 4.5-5 Results Generated from PL4UP, ISE and PLWAP

Algorithms	Frequent Sequence Results
ISE	{10},{20},{50},{60},{70},{80} {10,50},{10,60},{10,70},{10,80},{10,10},{10,20},{20,10} {10,20,10}
PLWAP	{10},{10,10},{10,20},{10,20,10},{10,50},{10,60},{10,70},{10,80} {20},{20,10} {50} {60} {70} {80}
EPL4UP	{10},{20},{50},{60},{70},{80} {10,10},{10,20},{20,10},{10,50},{10,60},{10,70},{10,80} {10,20,10}

Figure 4.5-6 Results Generated from EPL4UP, ISE and PLWAP

Algorithms	Frequent Sequence Results
PLWAP	{10},{10,10},{10,10,30},{10,20},{10,20,10},{10,20,10,30}, {10,20,30},{10,30} {20},{20,10},{20,10,30},{20,30} {30}
PL4UP_del	{10},{20},{30} {10,10},{10,20},{10,30},{20,10},{20,30} {10,10,30},{10,20,10}{10,20,30},{20,10,30} {10,20,10,30}

Figure 4.5-7 Results Generated from PL4UP_del and PLWAP

5. Conclusions and Future Work

ISE [MPT00b] or similar apriori-like algorithms generate numerous candidate itemsets that need to be computed at each level and scans the updated database U many times. For all types of database updates, ISE updates the old generated frequent patterns by mining data level-wise the same way.

If the PLWAP algorithm is directly used to mine the updated database, it has to scan the updated database U two times to construct PLWAP tree, and mine the tree step by step without using existing patterns and tree of the original database DB, which wastes a lot of reusable resources and is not wise.

The proposed PL4UP and EPL4UP inherit the advantages of PLWAP, and no huge candidate itemsets need to be generated. They also fully utilize old existing information like the old patterns and tree for mining the updated database U. Both of them use "Root set list" to update old patterns' counts in a small PLWAP tree of db instead of re-scanning db to update their count. They also use the new method " mask" to reduce the number of new mined patterns produced from the small PLWAP tree which need to be checked in old big PLWAP tree. These methods can save a lot of mining time. By using a formula of tolerance t, PL4UP can handle all cases without changing old big PLWAP tree to get higher efficiency. EPL4UP can handle a database that contains lots of previous small but now large items. It modifies and updates the old PLWAP tree, then applies the methods used in PL4UP to get good performance. Our experiments prove that both PL4UP and EPL4UP can achieve better performance than the apriori-like algorithm ISE and directly using algorithm PLWAP.

Future work should include how to apply the PL4UP and EPL4UP algorithm in an actual database like the student information database at the University of Windsor, or the web access log of the Computer Science School. Efficiently transforming the web log into a sequential database would be a hard task. Another area of interest is investigating the possibility of applying the proposal algorithms with some of the algorithms discussed in the literature to web content data.

References

[AMS+96] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In U. Fayyad and et al, editors, Advances in Knowledge Discovery and Data Mining. MIT Press, 1996.

[AS94] Rakesh Agrawal and Ranakrishnan Srikant . *Fast Algorithms for Mining Association Rules.* In Proc. of the 20th Int'l Conference on Very Large Databases, Santiago, Chile, September 1994

[AS95] Rakesh Agrawal and Ramakrishnan Srikant, *Mining Sequential Patterns,* Proc. of the 11th Int'l Conference on Data Engineering, Taipei, Taiwan, March 1995.

[BBA+99] A. Buchner, M. Baumgarten, S. Anand, M.Mulvenna, and J. Hughes. *Navigation pattern discovery from internet data.* In Proceedings of the WEBKDD'99 Workshop on web usage analysis and user profiling, pages 104-111, Melbourne Australia, August 24-28, 1998.

[Ber00] B. Berendt, *"Web usage mining, site semantics, and the support of navigation"* , In WEBKDD'00: Worskhop on Web Mining for E-Commerce -- Challenges and Opportunities, Boston, MA, USA, August 20, 2000.

[BL98] J. Borges and M. Levene. *Mining Association Rules in Hypertext Database*, In Proc. Of the Seventh, Int'l Conf. On User Modeling, Banff, Canada, 1998.

[BL99a] J. Borges and M. Levene. *Data mining of user navigation patterns* , In Proc. Of the WEBKDD'99 , San Diego, CA, USA, pp 31-36, 1999.

[BL99b] J. Borges and M. Levene. *"Heuristics for mining high quality user web navigation patterns"*, Research Note RN/99/68, Department of Computer Science, University College London, November 1999.

[CHN+96] D. Cheung, J. Han, V. Ng and C.Y. Wong, *"Maintenance of Discovered Association Rules in Large Databases: An Incremental Updating Technique"*, Proc. of 1996 Int'l Conf. on Data Engineering (ICDE'96), New Orleans, Louisiana, USA, Feb. 1996.

[CKL97] D. Cheung, B. Kao, and J. Lee. *"Discovering User Access Patterns on the World Wide Web"*, Proceedings of the 1st Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'97), Feb. 1997.

[CKR98] E. Cohen, B. Krishnamurthy, and J. Rexford. *Improving end-to-end performance of the web using server volumes and proxy filters.* In Proc. ACM SIGCOMM, pp 241-253, 1998.

[CMS97] R. Cooley, B. Mobasher, and J. Srivastava. *"Web Mining: Information and Pattern Discovery on the World Wide Web"*, in Proceedings of the 9th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'97), November 1997.

[CMS99] R. Cooley, B. Mobasher, and J. Srivastava. *"Data Preparation for Mining World Wide Web Browsing Patterns"*, in the Journal of Knowledge and Information Systems, Vol. 1, No. 1, 1999.

[CPY96] M.-S. Chen, J.-S. Park, and P. S. Yu. *Data Mining for Path Traversal Patterns in a Web Environment.* Proceedings of the 16th International Conference on Distributed Computing Systems, 385-392, May 27-30 1996.

[CPY98] M.S. Chen, J.S. Park, and P.S. Yu. *"Efficient Data Mining for Path Traversal Patterns"*, IEEE Trans. on Knowledge and Data Engineering, Vol. 10, No. 2, pp. 209-221, April 1998.

[ES02] Ezeife, C.I. and Yue Su, "Mining Incremental Association Rules with Generalized FP-tree", Proceedings of the fifteenth Canadian Conference on Artificial Intelligence, AI 2002, Calgary, Canada, published in Lecture Notes in Computer Science (LNCS) by Springer Verlag. May 25-29, 2002.

[GRS+99] M.N.Garofalakis, R.Rastogi, S.Seshadri, and K. Shim. *"Data Mining and the Web: Past, Present and Future"*, Proceeding of WIDM'99, Kansas City, Missouri, Nov, 1999.

[HK00] J. Han and M. Kamber *Data Mining: Concepts and Techniques* Morgan Kaufmann, 2000.

[HPM+00] J. Han, J. Pei, B. Mortazave-Asl, Q. Chen, U. Dayal, and M.-C. Hsu. *Free Span: Frequent Pattern-Projected Sequential Pattern Mining.* In Proc. 2000 Int. Conf. on Knowledge Discovery and Data Mining (KDD'00), Boston, MA, pp.355-359, August 2000.

[HPY00] J. Han, J. Pei and Y. Yin. *Mining Frequent Patterns without Candidate Generation. Proc. 2000 ACM-SIGMOD Int. Conf. on Management of Data (SIGMOD'00)*, pages 1-12, Dallas, TX, May 2000.

[JFM97] T. Joachims, D. Freitag, T. Mitchell, *"WebWatcher: A Tour Guide for the World Wide Web"*, Proceedings of IJCAI97, p770-775, Nagoya, Japan, August 1997

[JJK99] A.Joshi, K. Joshi, and R.Krishnapuram. *"On Mining Web Access Logs"*, Technical Report, CSEE Department, 205-210, UMBC, 1999.

[KB00] R. Kosala and H. Blockeel. *Web mining research: A survey,* ACM SIGKDD Explorations, 2(1), 1-15, 2000.

[LE03] Y. Lu and C.I. Ezeife. *"Position Coded Pre-Order Linked WAP-Tree for Web Log Sequential Pattern Mining"*, In Proceedings of The 7th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2003), Seoul, Korea, Apr. 30-May 2 2003.

[LS98] W. Lee and S. J. Stolfo. *"Data mining approaches for intrusion detection"*, In Proceedings of the 1998 USENIX Security Symposium, 50-56, 1998.

[MBN+99] S.K. Madria, S.S. Bhowmick, W.K. Ng, and E.P. Lim. *"Research issues in web data mining"*, In Proceedings of Data Warehousing and Knowledge Discovery, First International Conference, DaWak'99, 303-312, 1999.

[MCP98] F. Masseglia, F. Cathala and P. Poncelet. *PSP: Prefix Tree For Sequential Patterns,* Proceedings of the 2nd European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD'98), Nantes, France, LNAI, Vol. 1510, pp. 176-184, September 1998.

[MCS00] B. Mobasher, R. Cooley, and J. Srivastava. *"Automatic Personalization Based On Web Usage Mining"*, Communication of ACM, August 2000 (Volume 43, Issue 8).

[MDL+00a] B. Mobasher, H. Dai, T. Luo, Y. Sung, M. Nakagawa, and J. Wiltshire. *"Discovery of Aggregate Usage Profiles for Web Personalization"*, in Proceedings of the Web Mining for E-Commerce Workshop (WebKDD'2000), held in conjunction with the ACM-SIGKDD Conference on Knowledge Discovery in Databases (KDD'2000), August 2000, Boston.

[MDL+00b] B. Mobasher, H. Dai, T. Luo, Y. Sung, and J. Zhu. *"Integrating Web Usage and Content Mining for More Effective Personalization"*, in Proceedings of the International Conference on E-Commerce and Web Technologies (ECWeb2000), September 2000, Greenwich,UK.

[MJH+97] B. Mobasher, N. Jain, E. Han, and J. Srivastava. *"Web Mining: Pattern Discovery from World Wide Web Transactions"*, In Proceedings of the 9th IEEE International Conference on Tools with AI (ICTAI,97), Nov. 1997.

[MPT00a] F. Masseglia, P. Poncelet, and M. Teisseire. *"Web Usage Mining: How to Efficiently Manage New transactions and New Customers"*. Rapport de Recherche LIRMM, 18 pages, Fevrier 2000. Version courte dans Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery (PKDD'00), Lyon, France, September 2000.

[MPT00b] F. Masseglia, P. Poncelet and M. Teisseire, *"Incremental Mining of Sequential Patterns in Large Databases"*, Actes des 16imes Journes Bases de Donnes Avances (BDA'00), Blois, France, October 2000

[NM00] A. Nanopoulos and Y. Manolopoulos, *"Finding Generalized Path Patterns for Web Log Data Mining"*, in Proc. East-European Conference on Advanced Databases and Information Systems (ADBIS'00), 2000, pp.215-228.

[NM01] A. Nanopoulos and Y. Manolopoulos, *"Mining Patterns from Graph Traversals"*, Data and Knowledge Engineering, Vol. 37, No. 3, pp. 243-266, Jun. 2001

[PCY97] J.S. Park, M.S. Chen and P. S. Yu, *"Using a Hash-Based Method with Transaction Trimming for Mining Association Rules"*, IEEE Trans. on Knowledge and Data Engineering, Vol. 9, No. 5, pp. 813-825, October 1997.

[PE97] M. Perkowitz and O. Etzioni. *"Adaptive web sites: Automatically learning from user access patterns"*. In Proceedings of the Sixth Int. WWW Conference, Santa Clara, CA, 1997.

[PHM+00] Jian Pei, Jiawei Han, Behzad Mortazavi-asl, and Hua Zhu. *Mining Access Patterns Efficiently from Web Logs.* In Proc. 2000 Pacific-Asia Conf. On Knowledge Discovery and Data Mining (PAKDD'00), Kyoto, Japan, April 2000.

[PHM00] J. Pei, J. Han, and R. Mao. *CLOSET: An efficient algorithm for mining frequent closed itemsets*. In Proc. 2000 ACMSIGMOD Int. Workshop Data Mining and Knowledge Discovery (DMKD'00), 11-20, Dallas, TX, May 2000.

[PHM+01] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal and M-C. Hsu. *"PrefixSpan: Mining Sequential Patterns Efficiently by PrefixProjected Pattern Growth"*. In. Proc. 2001 Int. Conf. Data Engineering (ICDE'01), 215-224, Heidelberg, Germany, April 2001.

[PZO+99] S. Parthasarathy, M. J. Zaki, M. Ogihara and S. Dwarkadas, *"Incremental and Interactive Sequence Mining"*, In Proc.(1999) of the 8th International

Conference on Information and Knowledge Management (CIKM99), 251-258, Kansas City, MO, November 1999

[SA95] Ramakrishnan Srikant and Rakesh Agrawal. *Mining generalized association rules.* In 21st Int'l Conf. on Very Large Databases (VLDB), Zurich, Switzerland, Sept. 1995.

[SA96] R. Srikant and R. Agrawal, *Mining Sequential Patterns: Generalizations and Performance Improvements,* Proc. of the Fifth Int'l Conference on Extending Database Technology (EDBT). Avignon, France (1996).

[SCD+00] J.Srivastava, R.Cooley, M. Deshpande, and P.Tan, *Web Usage Mining: Discovery and Applications of Usage Patterns from Web Data,* SIGKDD Explorations, 1(2), 12-23, 2000.

[SF98] M. Spiliopoulou and L.C. Faulstich. *"WUM: A Tool for Web Utilization Analysis"*, In extended version of Proc. EDBT Workshop WebDB'98, LNCS 1590, pages 184-203. Springer Verlag, 1998.

[Spi99a] M. Spiliopoulou. *"The laborious way from data mining to web mining".* Int. Journal of Comp. Sys., Sci. & Eng., Special Issue on "Semantics of the Web", 14:113-126, Mar. 1999

[Spi99b] M. Spiliopoulou. *"Managing interesting rules in sequence mining".* In Poster proceedings of the 3rd European Conf. on Principles and Practice of Knowledge Discovery in Databases PKDD'99, number 1704 in LNAI, pages 554-560, Prague, Czech Republic, Sept. 1999. Springer Verlag.

[SY01] R. Srikant, Y. Yang. *"Ming web logs to improve website organization", in Proc. of the Tenth International World Wide Web Conference, Hong Kong, May 2001.*

[Zak00] M. J. Zaki, *"SPADE: An Efficient Algorithm for Mining Frequent Sequences"*, In Proc. of Machine Learning Journal, special issue on Unsupervised Learning (Doug Fisher, ed.), Vol. 42 Nos. 1/2, pages 31-60, Jan/Feb 2001

[Zai01] Osmar R. Zaïane, *"Web Usage Mining for a Better Web-Based Learning Environment"*, In Proc. Of Conference on Advanced Technology for Education, Banff, Alberta, June 27-28, 2001.

[ZE01] Zhou, Z., Ezeife, C.I., *"A Low-scan incremental association rule maintenance method"* Proceedings of the 14th Canadial Conference on Artificial Intelligence, AI 2001, holding June 7 to June 9, 2001, Ottawa, Canada.

Appendix A: Condensed ISE Algorithm

```
#include <iostream>
#include <map>
#include <fstream>
#include <list>
#include <deque>
#include <time.h>
#include <cmath>
#include <iomanip>
#include <conio.h>
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))


using namespace std;

struct candidate{
      deque<int> sequence;
      int count;
};

struct inform{
            int seqLength;
            int totalSeq;
            list<candidate>  itemSet;
};

typedef multimap<int, int, less<int> > sequence;
double frequency,upFreq;
double minSupp,oldTotalSeq,rowInDB;
bool finish,finalFinish;

char *sourceOrig="case3orig.data";
char *sourceChanged="case3insert2.data";
char *sourceUpdated="updated.data";
char *orig_result_ISE="total_origInfCase3Sol1S.data";
char *updated_large_ISE="updated_large_ISE.data";

int runTime = 0,runTime2=0;
const int an=30;
list<candidate> tempSeq[an], oldSeq[an],combineSeq[an],oldCand[an],remainedOldSeq[an],changedSmall[an];
list<candidate> updatedLarge[an],largeToLarge[an],largeToSmall[an];
list<inform> allInf;
list<inform> oldInf[an];
void ISE();
list<candidate> commonSeq(list<candidate> & , list<candidate> &,list<candidate>);
list<candidate> diffSeq(list<candidate> &,list<candidate> );
list<candidate> remainedSeq(list<candidate>&,list<candidate>,list<candidate>);
void countOldSeq(ifstream & ,bool);
list<candidate> Test(list<candidate>);
list<candidate> testFinal(list<candidate>);
list<candidate> countSeq(ifstream & , list<candidate> );
list<candidate> pruneSeq(list<candidate> , int ,bool ,list<candidate>&);
list<candidate> genSeq(list<candidate>,list<candidate> );
list<inform> recordTotalResult(list<candidate>, int , list<inform> );
void writeTotalResult(ofstream &, list<inform>);
void writeResult(list<candidate>, ofstream&,bool ) ;
void readResult(ifstream &, list<inform>&,list<candidate>&);//  int,int,
void readDatabase(ifstream &,double &,sequence &,sequence &);
list<candidate> pruneList1(list<candidate>,int);
sequence pruneSequence1(sequence,int);
sequence writeBackSeq(list<candidate> combineSeq);
list<candidate> commonPart(list<candidate> &, list<candidate> &,list<candidate> );
list<candidate> diffCommonSeq(list<candidate> &updatedL,list<candidate>oldS,list<candidate>remainedN);

int main()
{
            cout << "please enter the min support:";
```

```
	cin >> minSupp;

	ofstream result("changed_large_ISE.data", ios::app);
	 result.close();
	 ofstream result1("updated_large_ISE.data", ios::trunc);
	 result1.close();

	 int tim1 = time(0);

	 cout<<"Now start the program.....\n\n"<<endl;

	 ISE();

	 int tim2 = time(0);

	 cout<<"\n\nbegin time: "<<tim1<<'\n';
	 cout<<"end time  : "<<tim2<<'\n';
	 cout<<"The execution time is: "<< tim2-tim1;
	 cout<<"\n\nEnd the program"<<endl;

	return 0;
}

void ISE()
{
	sequence seq, duplicate;
	finish = false;
	finalFinish=false;

	ifstream read(orig_result_ISE, ios::in);//".data"

	double oldT;

	if(!read)
	{
		cerr<<"File could not be opened(orig_result_ISE)\n";
		exit(1);
	}
	cout<<"Now reading orig. information..";

	read>>oldT;
	oldTotalSeq=oldT;
	read>>rowInDB;
	int m=0;
	while(read&&!read.eof())//while read
	{
		readResult(read, oldInf[m],oldSeq[m]);
		m++;
	}

	//Next is Scan the changed part
	ifstream ins ( sourceChanged, ios::in);
	if ( !ins) {
		cerr << " File could not be opened(sourceChanged)(insert)\n";
		exit(1);
	}

	double seqNumber = 0;
	cout<<"Scan the changed database once to find the 1-sequence and update OS" << endl;

	while (ins && !ins.eof())
	{
		readDatabase(ins,seqNumber,duplicate,seq);

	}

	//		cout<<"end while and get candidate-1...\n";
	frequency = ceil(minSupp* seqNumber);
	upFreq=ceil(minSupp*(seqNumber+rowInDB));
```

```
//record all candidate-1 items of changed part
sequence::iterator i, bi;//,ti
for(i = seq.begin(); i != seq.end(); i++ )
{
        struct candidate newCand;
        newCand.sequence.push_front(i->first);
        newCand.count = i->second;//cout<<" e:"<<i->first;
        tempSeq[0].push_back(newCand);
}
oldCand[0]=tempSeq[0];
//combining old candidate-1 with changed candidate-1
int     k=0;
cout<<"\nCombining oldCand-1 with changedCand-1.."<<endl;

//combining the common candidate-1 and its'count
//then add difference of combined sequences

combineSeq[k]=commonSeq(oldSeq[0],tempSeq[0],combineSeq[k]);
combineSeq[k]=diffSeq(oldSeq[0],combineSeq[k]);
combineSeq[k]=diffSeq(tempSeq[0],combineSeq[k]);

cout<<"\nEnd Combining Candidate-1.\n";

combineSeq[1]=combineSeq[0];
updatedLarge[0]=combineSeq[0];

k=1;

//pruning to generate combined large-1 sequence.
combineSeq[k]=pruneList1(combineSeq[k],upFreq);

updatedLarge[k]=combineSeq[k];

//output combined large-1 sequence to file.
cout<<"\nFound updated 1-sequence .....\n"<<"k:"<<k<<endl;

ofstream result1(updated_large_ISE, ios::app);//".data"
bool writeCount=true;
writeResult(combineSeq[k],result1,writeCount);

//generating large-1 sequence for insert part
list<candidate> tempCombineSeq=combineSeq[1];
oldCand[1]=commonPart(oldCand[0],tempCombineSeq,oldCand[1]);

seq.clear();
seq=writeBackSeq(oldCand[1]);
oldCand[1].clear();

for(i = seq.begin(); i != seq.end(); i++ )
{
        struct candidate newCand;
        newCand.sequence.push_front(i->first);
         newCand.count = i->second;//cout<<" e:"<<i->first;
         oldCand[1].push_back(newCand);
}

if(seq.size()==0)
        cout<<"Cannot find 1-sequence in changed part,please lower min sup.\n";
else//else 1 begin.
{
        cout<<"Found 1-sequence .....\n";

        list<candidate> tempCombine,tempOldSeq;

        tempCombine=combineSeq[1];
        tempOldSeq=oldSeq[1];

        largeToLarge[1]=commonPart(oldSeq[1],combineSeq[1],largeToLarge[1]);
        largeToSmall[1]=diffSeq(oldSeq[1],largeToSmall[1]);
```

```
combineSeq[1]=tempCombine;
oldSeq[1]=tempOldSeq;

//check  remained old seq. from level 1(L1) to L(oldTotalSeq)
for(int ii=2; ii<=oldTotalSeq; ii++)
{
        remainedOldSeq[ii]=remainedSeq(oldSeq[ii],largeToSmall[1],remainedOldSeq[ii]);
}

//update oldSeq[ii];
ifstream insert(sourceChanged,ios::in);
countOldSeq(insert,false);
cout<<" end update old seq:"<<endl;
for( ii=2; ii<=oldTotalSeq; ii++)
{
  oldSeq[ii]=pruneSeq(oldSeq[ii],upFreq,false, changedSmall[runTime2]);
}

// Next is generate 2-sequence
cout<<"\nGenerating 2-seq of inserted db..\n";
k=1;
list<candidate>::iterator ui,bui;
ui=bui=oldCand[k].begin();

while(ui != oldCand[k].end())
{
        deque<int>::iterator i,bi;
        for(i=ui->sequence.begin(); i != ui->sequence.end(); i++)
        {
                for(bui=oldCand[k].begin(); bui !=oldCand[k].end(); bui++)
                {
                        for(bi=bui->sequence.begin(); bi != bui->sequence.end(); bi++)
                        {
                                struct candidate newCand;
                                newCand.count = 0;
                                newCand.sequence.push_front(*i);
                                newCand.sequence.push_back(*bi);
                                oldCand[k+1].push_back(newCand);
                        }
                }

        }
        ui++;
}
cout<<"\nFinish to Generate 2-seq of inserted db..\n";

runTime = 2;

if(oldCand[2].size() !=0)//candList
{
        while(runTime< oldTotalSeq && !finish)
        {
                oldCand[runTime] = Test(oldCand[runTime]);//candListcandList
        }
}

cout<<"Finish Testing orig "<<runTime<<"-sequence..."<<endl;
cout<<"Begine to test updated part.k= "<<runTime<<"-sequence..."<<endl;
 while(finalFinish==false)
{
        oldCand[runTime]= testFinal(oldCand[runTime]);
}
}//end else 1


cout<<"\nOutput all information(candidat-1, L1,2..) of updated U:\n";
ofstream result2("totalInf_ISE.data", ios::trunc);
result2<<runTime-1<<"\t"<<seqNumber+rowInDB<<endl;
for(int h=0; h<runTime; h++)
```

```
		{
			allInf=recordTotalResult(updatedLarge[h],h,allInf);
		}
		writeTotalResult(result2,allInf);

	return;

}//end ISE

//===========below is the functions==============
//extracting common part of 2 sequence and add two counts into common
list<candidate> commonSeq(list<candidate> &oldSeq, list<candidate> &tempSeq,list<candidate> combineSeq)
{
		list<candidate>::iterator itemBrow,oldTemp, changedBrow, changedTemp;
		itemBrow=oldSeq.begin();
		changedBrow=tempSeq.begin();
		bool compareF = false;
		struct candidate newCand;

		while(itemBrow != oldSeq.end() && compareF != true)//common
		{
			bool matchFlag=false;
			int jj=0, kk=0;
			deque<int> infSeq=itemBrow->sequence;
			int infCount=itemBrow->count;
			deque<int> changedSeq=changedBrow->sequence;
			int changedCount=changedBrow->count;

			while(jj<infSeq.size()&& kk<changedSeq.size()&&!matchFlag)
			{
				if(infSeq[jj]==changedSeq[kk])
				{
					jj++;
					if(jj==infSeq.size())
					matchFlag=true;
				}
				kk++;
			}

			if(matchFlag==true)
			{
				newCand.sequence=infSeq;
				newCand.count = infCount+changedCount;
				combineSeq.push_back(newCand);

				oldTemp=itemBrow;
				changedTemp=changedBrow;
				itemBrow++;
				changedBrow++;
				oldSeq.erase(oldTemp);
				tempSeq.erase(changedTemp);

				changedBrow=tempSeq.begin();

				//if(changedBrow != tempSeq.end())
				//	changedBrow=tempSeq.begin();
				//else
				//	compareF=true;
			}
			else if (changedBrow != tempSeq.end())
					changedBrow++;
				else if(itemBrow != oldSeq.end())
				{
					itemBrow++;
					changedBrow=tempSeq.begin();
				}
					else
					compareF=true;
		}//end while common
			return combineSeq;
```

```
}
//end of commonSeq


//appending differece part of 2 sequence
list<candidate> diffSeq(list<candidate> &tempSeq,list<candidate> combineSeq)
{
	struct candidate newCand;
	list<candidate>::iterator itemBrow, changedBrow;

	for(changedBrow=tempSeq.begin(); changedBrow != tempSeq.end(); changedBrow++)
	{
		deque<int> changedSeq=changedBrow->sequence;
		int changedCount=changedBrow->count;
		newCand.sequence=changedSeq;
		newCand.count = changedCount;
		combineSeq.push_back(newCand);
	}
	return combineSeq;
}
//end of diffSeq


//using largeToSmall-1 to check all oldSeq and erase them
list<candidate> remainedSeq(list<candidate> &oldSeq,list<candidate>largeToSmall,list<candidate>remainedOldSeq)
{
	list<candidate>::iterator itemBrow,oldTemp, changedBrow, changedTemp;
	//	list<candidate>::iterator oldTemp, changedTemp;
	itemBrow=oldSeq.begin();
	changedBrow=largeToSmall.begin();
	bool compareF = false;
	struct candidate newCand;
	//	int tm1=0,tm2=0,tm3=0;
	while( compareF != true)//common//itemBrow != oldSeq.end() &&
	{
		bool matchFlag=false;
		int jj=0, kk=0;

		deque<int> infSeq=itemBrow->sequence;
		int infCount=itemBrow->count;
		deque<int> changedSeq=changedBrow->sequence;
		int changedCount=changedBrow->count;

		while(jj<infSeq.size()&& kk<changedSeq.size()&&!matchFlag)
		{
			if(infSeq[jj]==changedSeq[kk])
			{
				kk++;
				if(kk==changedSeq.size())
				matchFlag=true;
			}
			jj++;
		}

		if(matchFlag==true)
		{
			itemBrow=oldSeq.erase(itemBrow);
			if(itemBrow != oldSeq.end())
				itemBrow++;
			else
			{
				if(changedBrow != largeToSmall.end())
				{
					changedBrow++;
					itemBrow=oldSeq.begin();
				}
				else
					compareF=true;
			}
		}
```

```
                else
                {
                        if (itemBrow != oldSeq.end())
                        {itemBrow++;}
                        else
                        {
                                if(changedBrow != largeToSmall.end())
                                {
                                changedBrow++;
                                itemBrow=oldSeq.begin();
                                }
                                else
                                compareF=true;
                        }
                }
        }//end while common

          remainedOldSeq=oldSeq;
                return remainedOldSeq;
}//end of remainedSeq

void countOldSeq(ifstream & inFile, bool reduceC)
{
        if ( !inFile)
        {
                cerr << " File could not be opened(countOldSeq)\n";
                exit(1);
        }

        int event, cid, number;

        while (inFile && !inFile.eof())
        {
                deque<int> inSequence;
                inFile >> cid;
                inFile >> number;

                for (int i = 0; i < number; i++)
                 {
                        inFile >> event;
                        inSequence.push_back(event);
                }

                //accumulating support of candidates of candList in changed part or updated part
                for(int ii=2; ii<=oldTotalSeq; ii++)
                {
                        for ( list<candidate>::iterator candBrow = oldSeq[ii].begin(); candBrow != oldSeq[ii].end(); candBrow++)
                        {
                                deque<int> candSequence = candBrow->sequence;
                                unsigned int j = 0, k = 0;
                                bool find = false;

                                while (  j < candSequence.size() && k < inSequence.size() && !find )
                                 {
                                        if ( candSequence[j] == inSequence[k])
                                        {
                                                j++;
                                                if ( j == candSequence.size())
                                                 find = true;
                                        }
                                        k++;
                                }
                                if ( find )
                                {
                                        if(reduceC ==true)
                                                candBrow->count--;
                                        else
                                                candBrow->count ++;
                                }
                        }
```

```
                    }
            }
            return ;
}//end of countSeq

// testing seqeunce is frequent or not
list<candidate> Test(list<candidate> candList)
{
            cout<<"Testing "<<runTime<<"-sequence..oldSeq["<<runTime<<"]="<<oldSeq[runTime].size()<<endl;
            ifstream insert(sourceChanged,ios::in);
            candList=countSeq(insert,candList);

            list<candidate>remainedNewSeq;
            candList=pruneSeq(candList,frequency,false, changedSmall[runTime]);
            remainedNewSeq=candList;
            cout<<"finish checking. remainedNewSeq="<<remainedNewSeq.size()<<",begin to scan DB.\n";

            ifstream inFile ( sourceOrig, ios::in);
            remainedNewSeq=countSeq(inFile, remainedNewSeq);
            inFile.close();
            cout<<"finish scanning DB.\n";
            remainedNewSeq=pruneSeq(remainedNewSeq,upFreq,false, changedSmall[runTime]);
            if(remainedNewSeq.size()<=1)
                        candList=remainedNewSeq;
            cout<<"finish pruning remainedNewSeq="<<remainedNewSeq.size()<<".\n";

            updatedLarge[runTime]=diffCommonSeq(remainedNewSeq,oldSeq[runTime],updatedLarge[runTime]);
            updatedLarge[runTime]=diffSeq(oldSeq[runTime],updatedLarge[runTime]);

            //write updated result into file
            ofstream result1(updated_large_ISE, ios::app);
            bool writeCount=true;
            writeResult(updatedLarge[runTime],  result1,writeCount);
            result1.close();

            cout<<"Finish testing "<<runTime<<"-sequence. and This is result..."<<endl;

   list<candidate> newCandList;
   if( updatedLarge[runTime].size() == 0 )
            finish = true;
   else
   {
            runTime++;
            cout<<"\nGenerating "<<runTime<<"-sequence....."<<endl;
            newCandList=genSeq(candList,newCandList);
   }

            return newCandList;
}//end Test

//testFinal begin
list<candidate> testFinal(list<candidate> combineS)
{
            // testing updated part's seqeunce is frequent or not
            cout<<"Testing updated "<<runTime<<"-sequence..."<<endl;

            bool smallFlag=true;
            //counting support then pruning candiates in updated part
            ifstream readFile ( sourceUpdated, ios::in);
            combineSeq[runTime]=combineS;
            combineSeq[runTime]=countSeq(readFile, combineSeq[runTime]);

            smallFlag=false;
            combineSeq[runTime]=pruneSeq(combineSeq[runTime],upFreq,smallFlag, changedSmall[runTime]);
            updatedLarge[runTime]=combineSeq[runTime];

            cout<<"Finish updated testing "<<runTime<<"-sequence. and This is result..."<<endl;

            ofstream result1(updated_large_ISE, ios::app);
            bool writeCount=true;
```

```
        writeResult(updatedLarge[runTime],  result1,writeCount);

        //checking stop running condition
         list<candidate> newCandList;
         if(updatedLarge[runTime].size() == 0 )
        {
                finalFinish = true;
                newCandList=updatedLarge[runTime];
        }
         else
         {
                runTime++;
                 cout<<"\nGenerating updated "<<runTime<<"-sequence....."<<endl;
                newCandList=genSeq(updatedLarge[runTime-1], newCandList);//combineSeq[runTime-1]
        }

                return newCandList;

}//testFinal end

list<candidate> countSeq(ifstream & inFile, list<candidate> candList)
{
        if ( !inFile)
        {
                cerr << " File could not be opened(countSeq)\n";
                exit(1);
        }

         int event, cid, number;
        int No=0;
        while (inFile && !inFile.eof())
        {
                deque<int> inSequence;
                inFile >> cid;
                inFile >> number;

                for (int i = 0; i < number; i++)
                 {
                        inFile >> event;
                         inSequence.push_back(event);
                 }

                //accumulating support of candidates of candList in changed part or updated part
                for ( list<candidate>::iterator candBrow = candList.begin(); candBrow != candList.end(); candBrow++)
                 {
                        deque<int> candSequence = candBrow->sequence;
                        unsigned int j = 0, k = 0;
                        bool find = false;

                         while (  j < candSequence.size() && k < inSequence.size() && !find )
                         {
                                if ( candSequence[j] == inSequence[k])
                                {
                                        j++;
                                        if ( j == candSequence.size())
                                        find = true;
                                }
                                k++;
                         }
                        if ( find )
                                candBrow->count ++;
                }
        }
        return candList;
}//end countSeq

//pruning seq.
list<candidate> pruneSeq(list<candidate> candList, int frequency,bool smallF,list<candidate>& changedSmall)
{
        list<candidate>::iterator candBrow = candList.begin();
```

```
		while( candBrow != candList.end() )
		{
			if ( candBrow->count < frequency )
			{
				if(smallF=true)
				changedSmall.push_back(*candBrow);//get small seq
				candBrow = candList.erase(candBrow);
			}
			else candBrow++;
		}
			return candList;
}

//gen seq
list<candidate> genSeq(list<candidate> candList,list<candidate> newCandList)
{
		list<candidate>::iterator oneBrowser, twoBrowser;
		for ( oneBrowser = candList.begin(); oneBrowser != candList.end(); oneBrowser++)
		{
			deque<int> firstSeq = oneBrowser->sequence;
			for ( twoBrowser = candList.begin(); twoBrowser != candList.end(); twoBrowser++)
			{
				deque<int> secondSeq = twoBrowser->sequence;
				if( firstSeq[1] == secondSeq[0])
				{
					int j = 1;
					int length = firstSeq.size();
					bool match = true;
					while( j < length-1 && match )
					{
						if (firstSeq[j+1] != secondSeq[j])
						match = false;
						 j++;
					 }

					if (match)
					{
						struct candidate newCand;
						secondSeq.push_front(firstSeq.front());
						newCand.sequence = secondSeq;
						newCand.count = 0;
						newCandList.push_back(newCand);
					}
				}
			}

  }

		return newCandList;
}

//pruning list seq1
list<candidate> pruneList1(list<candidate>combineSeq,int upFreq)
{
		list<candidate>::iterator ui, bui;
		ui=bui=combineSeq.begin();
		while(ui != combineSeq.end())
		{
			if(ui->count < upFreq)
			{
				bui++; combineSeq.erase(ui); ui=bui;
			}
			else
			{
				if(bui != combineSeq.end())
				{
					ui++; bui++;
				}
				else
```

```
                                        ui=bui;
                }
        }
        return combineSeq;
}

//pruning sequence seq1
sequence pruneSequence1(sequence seq,int frequency)
{
        sequence::iterator i, bi;
        bi=i=seq.begin();

        while(i != seq.end())
        {
                if(i->second < frequency)
                {
                        bi++; seq.erase(i); i=bi;
                }
                else
                {
                        if(bi != seq.end())
                        {
                                i++; bi++;
                        }
                        else
                                i=bi;
                }
        }
        return seq;
}

//obtain common part from 2 seq, but not add count
list<candidate> commonPart(list<candidate>& oldSeq, list<candidate>& tempSeq,list<candidate> combineSeq)
{
        list<candidate>::iterator itemBrow,oldTemp, changedBrow, changedTemp;
        itemBrow=oldSeq.begin();
        changedBrow=tempSeq.begin();
        bool compareF = false;
        struct candidate newCand;

        while(itemBrow != oldSeq.end() && compareF != true)//common
        {
                bool matchFlag=false;
                int jj=0, kk=0;

                deque<int> infSeq=itemBrow->sequence;
                int infCount=itemBrow->count;
                deque<int> changedSeq=changedBrow->sequence;
                int changedCount=changedBrow->count;

                while(jj<infSeq.size()&& kk<changedSeq.size()&&!matchFlag)
                {
                        if(infSeq[jj]==changedSeq[kk])
                        {
                                jj++;
                                if(jj==infSeq.size())
                                matchFlag=true;
                        }
                        kk++;
                }

                if(matchFlag==true)
                {
                        newCand.sequence=infSeq;
                        newCand.count = changedCount;
                        combineSeq.push_back(newCand);

                        oldTemp=itemBrow;
                        changedTemp=changedBrow;
                        itemBrow++;
```

```
                        changedBrow++;
                        oldSeq.erase(oldTemp);
                        tempSeq.erase(changedTemp);
                        changedBrow=tempSeq.begin();
                }
                else if (changedBrow != tempSeq.end())
                        changedBrow++;
                        else if(itemBrow != oldSeq.end())
                        {
                                itemBrow++;
                                changedBrow=tempSeq.begin();
                        }
                        else
                                compareF=true;
        }//end while common
                return combineSeq;
}//end of commonPart

list<candidate> diffCommonSeq(list<candidate> &updatedL,list<candidate>oldS,list<candidate>remainedN)
{
        list<candidate>::iterator firstBrow,firstTemp, secondBrow, secondTemp;

        firstBrow=updatedL.begin();
        secondBrow=oldS.begin();
        bool compareF = false;
        struct candidate newCand;

        while(compareF != true)//outer-while
        {
                bool matchFlag=false,move1=false,move2=false,record=false;
                int j1=0, k2=0;

                deque<int> firstSeq=firstBrow->sequence;
                int firstCount=firstBrow->count;
                deque<int> secondSeq=secondBrow->sequence;
                int secondCount=secondBrow->count;

                while(j1<firstSeq.size()&& k2<secondSeq.size()&&!matchFlag&&!move1&&!move2)
                {//inner-while
                        if(firstSeq[j1]==secondSeq[k2])
                        {
                                j1++;
                                if(j1==firstSeq.size())
                                matchFlag=true;

                                k2++;
                        }
                        else if(firstSeq[j1]<secondSeq[k2])
                        {
                                record=true;
                                move1=true;
                        }
                        else if(firstSeq[j1]>secondSeq[k2])
                        {
                                move2=true;
                        }

                }//inner-while

                if(record==true)
                {
                        newCand.sequence=firstSeq;
                        newCand.count = firstCount;
                        remainedN.push_back(newCand);
                }

                if(matchFlag==true)
                {
                        list<candidate>::iterator temp;
                        temp=firstBrow;
```

```
                    firstBrow++;
                    updatedL.erase(temp);
                    secondBrow++;
            }

            if(move1==true)
            {
                    firstBrow++;
            }
            if(move2==true)
            {
                    secondBrow++;
            }

            if (firstBrow == updatedL.end())
                    compareF=true;
            else if(secondBrow == oldS.end())
                    {
                            while(firstBrow != updatedL.end())
                            {
                                    firstSeq=firstBrow->sequence;
                                    firstCount=firstBrow->count;

                                    newCand.sequence=firstSeq;
                                    newCand.count = firstCount;
                                    remainedN.push_back(newCand);
                                    firstBrow++;
                            }
                            compareF=true;
                    }

    }//end outer-while
            return remainedN;
}//end diffCommonSeq
```

Appendix B: Condensed PLWAP Algorithm

```
#include <iostream>
#include <map>
#include <fstream>
#include <list>
#include <queue>
#include <time.h>
#include <conio.h>//for getch()
#include <cmath>
#define min(a, b) ((a) < (b) ? (a) : (b))

using namespace std;

struct positionCode
{
      unsigned int code;
      positionCode *next;
};

struct node
{
      int event;
      int occur;
      int pcLength;
      int CountSon;
      positionCode *pcCode;
      node *nextLink;
      node *lSon;
      node *rSibling;
      node *parent;
};

struct candidate
{
      queue<int> sequence;
      int count;
};

struct inform
{
      int seqLength;
      int totalSeq;
      list<candidate>  itemSet;
};

struct linkheader
{
      int event;
      int occur;
      node *link;
      node *lastLink;
};

typedef multimap<int, int, less<int> > sequence;
list<linkheader> lnkhdr;
node *root;
int frequency,inputFreq=4;
int seqNumber;
float minSupp;
int runTime = 1;
int maxTempPattern=0;
const int an=30;

char *sourceOrig="test.data";
char *total_origInf_W4UP="total_origInfForReport_W4UP.data";
char *result_PLWAP="result_origForReportPLWAP.data";
```

```
list<candidate> oldCand[an],tempSeq[an],combineSeq[an],updatedLarge[an],provedLarge[an],changedSmall[an];
list<candidate> tempS[an],combineS[an],updatedL[an],provedL[an],remainedOldS[an],changedS[an];
list<inform> allInf,oldInf[an],remainedInf;
list<inform> recordTotalResult(list<candidate>, int, list<inform>);
void writeTotalResult(ofstream &, list<inform>);
void BuildTree(char*);
void BuildLinkage(node *);
positionCode* makeCode(int, positionCode*, bool);
int checkPosition(positionCode*, int, positionCode*, int);
void MiningProcess(list<node*>, queue<int>, int );

int main()
{
    list<node*> newRootSet;
    queue<int> beginPattern;
    cout << "please enter the frequency:";
    cin >> minSupp;

    ofstream result(result_PLWAP, ios::trunc);
    result.close();

    int tim1 = time(0);

    cout<<"Now start the program.....\n\n"<<endl;
    BuildTree(sourceOrig);
    newRootSet.push_back(root);
    cout<<"\nBegin the mining process"<<endl;
    MiningProcess(newRootSet, beginPattern, seqNumber);

    int tim2 = time(0);

    cout<<"\n\nbegin time: "<<tim1<<'\n';
    cout<<"end time  : "<<tim2<<'\n';
    cout<<"The execution time is: "<< tim2-tim1<< endl;
    cout<<"\n\nEnd the program"<<endl;

    cout<<"Finish mining process,max length of seq: "<<maxTempPattern<<"-sequence..."<<endl;
    ofstream result2;

    allInf.clear();
    result2.open(total_origInf_W4UP, ios::trunc);
    result2<<maxTempPattern+1<<"\t"<<seqNumber<<endl;
    for(int h=0; h<maxTempPattern+1; h++)
    {
        allInf=recordTotalResult(updatedLarge[h],h,allInf);
    }
    writeTotalResult(result2,allInf);

    return 0;
}

void MiningProcess(list<node*> rootSet, queue<int> basePattern, int Count)
{
    list<linkheader>::iterator pnt;
    cout<<"\nEntering the mining process "<< runTime <<" times" <<endl;
    runTime++;

    for( pnt = lnkhdr.begin(); pnt != lnkhdr.end(); pnt++)
    {
        list<node*> newRootSet;
        node * SavePoint = NULL;
        bool DescSave = false;
        bool RootUsed = false;
        int totalSon = 0;
        int count = 0;
        int emptySon = 0;
        int RootCount = Count;
        node * linkBrow = pnt -> link;
        list<node*>::iterator rootBrow = rootSet.begin();
```

```
while (linkBrow != NULL && rootBrow != rootSet.end() && RootCount >= frequency )
{

    int check = checkPosition(((*rootBrow)->lSon)->pcCode, (*rootBrow)->pcLength+1, linkBrow->pcCode, linkBrow->pcLength);
    switch ( check )
    {
    case 0:
        if ( SavePoint != NULL )
        {
            if ( SavePoint->lSon == NULL )
                DescSave = false;
            else
                if (checkPosition( (SavePoint->lSon)->pcCode, SavePoint->pcLength+1, linkBrow->pcCode, linkBrow->pcLength)== 0)
                    DescSave = true;
                else DescSave = false;
        }

        if ( !DescSave)
        {
            count = count + linkBrow->occur;
            totalSon = totalSon + linkBrow->CountSon;
            RootUsed = true;
            SavePoint = linkBrow;
            if (linkBrow->lSon != NULL)
                newRootSet.push_back(linkBrow);
            else
                emptySon = emptySon + linkBrow->occur;
        }

        linkBrow = linkBrow->nextLink;
        break;
    case 1:
        if (!RootUsed)
            RootCount = RootCount - (*rootBrow)->occur;
        rootBrow++;
        RootUsed = false;
        break;
    case 2:
        linkBrow = linkBrow->nextLink;
        break;
    case 3:
        linkBrow = linkBrow->nextLink;
        break;
    }
}// end while

if ( count >= frequency)
{
    queue<int> tempPattern = basePattern;
    tempPattern.push(pnt->event);
    struct candidate newCand;
    newCand.sequence=tempPattern;
    newCand.count=count;
    updatedLarge[tempPattern.size()].push_back(newCand);

    if(maxTempPattern<tempPattern.size())
        maxTempPattern=tempPattern.size();

    queue<int> otherPattern = tempPattern;

    ofstream result(result_PLWAP, ios::app);

    while(!tempPattern.empty())
    {
        result<<tempPattern.front()<<"\t";
        tempPattern.pop();
    }
    result<<endl;

    list<node*>::iterator ii=newRootSet.begin();
```

```
                if ( totalSon >= frequency)
                {
                        MiningProcess(newRootSet, otherPattern, count-emptySon);
                }
            }
        }
        return;
}

void BuildTree(char *sourceFile)
{
    sequence seq, duplicate;
    //Next is Scan the database
    cout<<"Scan the database first time: " << runTime <<endl;
    ifstream ins ( sourceFile, ios::in);

    if ( !ins)
    {
        cerr << " File could not be opened\n";
        exit(1);
    }
    sequence::iterator point, eflag;
    int event, cid, number;
    seqNumber = 0;

    while (ins && !ins.eof())
    {
        ins >> cid;
        ins >> number;
        seqNumber++;
        duplicate.clear();

        for(int i=0; i< number; i++)
        {
            ins >> event;

            if ( duplicate.find(event) == duplicate.end())
            {
                duplicate.insert(sequence::value_type(event, 1));

                point = seq.find(event);
                eflag = seq.end();
                if ( point != eflag)
                        point ->second ++;
                else
                        seq.insert( sequence::value_type(event,1));

            }
        }
    }
    frequency =ceil((float) (minSupp* seqNumber));
    sequence::iterator i, bi;

    //record all candidate-1 items
    for(i = seq.begin(); i != seq.end(); i++ )
    {
        struct candidate newCand;
        newCand.sequence.push(i->first);
        newCand.count = i->second;
        updatedLarge[0].push_back(newCand);
    }

    int n=0;
    bi=i=seq.begin();
    while(i != seq.end())
    {
                if(i->second < frequency)
                {       cout<<"sc:"<<i->second;
                        bi++; seq.erase(i); i=bi; //n++;
                }
```

```
                else
                {
                        if(bi != seq.end())
                        {
                                i++; bi++;
                        }
                        else
                                i=bi;
                }
        }
                cout<<"end while .. ..\n";

cout<<"Finish scanning the database: " << runTime <<endl;
// Next is build the WAP tree

bool newLink;
node *Tranversal, *newNode, *Parent;

root = new node;
root->event = -1;
root->occur = seqNumber;
root->CountSon = 0;
root->pcLength = 0;
root->parent = NULL;
root->lSon = NULL;
root->rSibling = NULL;
root->nextLink = NULL;
root->pcCode = NULL;

ifstream inFile ( sourceFile, ios::in);

if ( !inFile)
{
       cerr << " File could not be opened\n";
       exit(1);
}

cout <<"Begin to build tree " << runTime << endl;
while (inFile && !inFile.eof())
{
      inFile >> cid;
      inFile >> number;

      Tranversal = root;

      for(int i=0; i< number; i++)
      {
          inFile >> event;

          if(seq.find(event) != seq.end())
          {
                Parent = Tranversal;

           if( Tranversal->lSon == NULL)
           {
                newNode = new node;
                newNode->event = event;
                newNode->occur = 1;
                newNode->lSon = NULL;
                newNode->rSibling = NULL;
                newNode->nextLink = NULL;
                newNode->CountSon = 0;
                Parent->CountSon ++;
                newNode->parent = Parent;
                newNode->pcLength = Tranversal->pcLength + 1;
                newNode->pcCode = makeCode(Tranversal->pcLength,Tranversal->pcCode,true);
                Tranversal->lSon = newNode;
                Tranversal = newNode;
           }
           else
```

```
                {
                    Tranversal = Tranversal->lSon;
                    if ( Tranversal->event == event)
                    {
                        (Tranversal->parent)->CountSon++;
                        Tranversal->occur++;
                    }
                    else
                    {
                        bool find= false;
                        while(Tranversal->rSibling != NULL && !find )
                        {
                            Tranversal = Tranversal->rSibling;
                            if ( Tranversal->event == event)
                            {
                                Tranversal->occur ++;
                                (Tranversal->parent)->CountSon++;
                                find = true;
                            }
                        }
                        if (!find)
                        {
                            newNode = new node;
                            newNode->event = event;
                            newNode->occur = 1;
                            newNode->lSon = NULL;
                            newNode->rSibling = NULL;
                            newNode->nextLink = NULL;
                            newNode->CountSon = 0;
                            Parent->CountSon ++;
                            newNode->parent = Parent;
                            newNode->pcLength = Tranversal->pcLength + 1;
                            newNode->pcCode = makeCode(Tranversal->pcLength,Tranversal->pcCode, false);
                            Tranversal->rSibling = newNode;
                            Tranversal = newNode;
                        }
                    }
                }
            }
        }
    }

    // next tranversing all tree to build the Pre-order linkage
    linkheader *newLinkHeader;
    for ( i = seq.begin(); i!=seq.end(); i++)
    {
        newLinkHeader = new linkheader;
        newLinkHeader->link = NULL;
        newLinkHeader->lastLink = NULL;
        newLinkHeader->event= i->first;
        newLinkHeader->occur= i->second;
        lnkhdr.push_back(*newLinkHeader);
        free(newLinkHeader);
    }
    cout<<"End of building tree and begin to build linkage..."<<endl;
    BuildLinkage(root->lSon);
    cout<<"End of building linkage...\n";

    return;
}


void BuildLinkage(node *start)
{
    if (start !=NULL)
    {
        list<linkheader>::iterator lnkBrow = lnkhdr.begin();
        while (lnkBrow->event != start->event && lnkBrow != lnkhdr.end())
            lnkBrow++;
```

```
            node *lastLinkage;
            lastLinkage = lnkBrow->lastLink;
            if (lastLinkage == NULL )
                lnkBrow->link = start;
            else
                lastLinkage->nextLink = start;
            lnkBrow->lastLink = start;

            BuildLinkage(start->lSon);
            BuildLinkage(start->rSibling);
        }
        else return;
}


positionCode * makeCode(int length, positionCode *pCode, bool addOne)
{
    positionCode *start, *browser, *newPC;
    int leftCount = length % 32;
    int linkCount = (int)(length / 32);

    if ( linkCount == 0 )
    {
        start = new positionCode;
        start->next = NULL;
        if (length == 0)
            start->code = 1 << 31;
        else
            if (addOne)
                start->code = pCode->code | (1<<(31-leftCount));
            else
                start->code = pCode->code ;
        return start;
    }
    else
    {
        newPC = new positionCode;
        newPC->code = pCode->code;
        pCode = pCode ->next;
        start = newPC;
        browser = start;

        for(int i = 1; i < linkCount; i++)
        {
            newPC = new positionCode;
            newPC->code = pCode->code;
            browser-> next = newPC;
            browser = newPC;
            pCode = pCode->next;
        }

        if (leftCount == 0)
        {
            newPC = new positionCode;
            if (addOne)
                newPC->code = 1 << 31;
            else
                newPC->code = 0 << 31;
            newPC->next = NULL;
            browser->next = newPC;
        }
        else
        {
            newPC = new positionCode;

            if (addOne)
                newPC->code = pCode->code | (1<<(31-leftCount));
            else
                newPC->code = pCode->code;
```

```
                newPC ->next = NULL;
                browser->next = newPC;
            }
        }

        return start;
}


int checkPosition(positionCode *ancestor, int aLength, positionCode *descendant, int dLength)
{
    if (aLength == 1 )
        return 0;

    int length = min(aLength,dLength);
    int linkCount = (int)(length / 32);
    int leftCount = length % 32;
    for( int i = 0; i < linkCount; i++)
    {
        if ( ancestor->code > descendant->code)
            return 1;
        else
            if ( ancestor->code < descendant->code)
                return 2;
            else
            {
                ancestor = ancestor->next;
                descendant = descendant->next;
            };
    }

    unsigned int aCode, dCode;
    if ( aLength <= dLength )
    {
        if (leftCount == 0)
            return 0;

        aCode = ancestor -> code >> ( 32 - leftCount );
        dCode = descendant -> code >> ( 32 - leftCount );
        if (aCode == dCode )
            return 0;
        else
            if (aCode < dCode)
                return 2;
            else return 1;
    }

    else
    {
        if (leftCount == 0)
            dCode = 1;
        else
            dCode = ( descendant -> code >> ( 31 - leftCount )) | 1 ;
        aCode = ancestor -> code >> ( 31 - leftCount );


        if ( aCode == dCode )
            return 3;
        else
            if (aCode < dCode)
                return 2;
            else return 1;
    }
}
```

Appendix C: Condensed PL4UP and EPL4UP Algorithms

```
#include <iostream>
#include <map>
#include <fstream>
#include <list>
#include <queue>
#include <time.h>
#include <cmath>
#include <iomanip>
#include <conio.h>
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))

using namespace std;

struct positionCode
{
    unsigned int code;
    positionCode *next;
};

struct node
{
    int event;
    int occur;
    int pcLength;
    int CountSon;
    positionCode *pcCode;
    node *nextLink;
    node *lSon;
    node *rSibling;
    node *parent;
};

struct linkheader
{
    int event;
    int occur;
    node *link;
    node *lastLink;
};

struct branchHeader
{
     int event;
     int occur;
     int branchL;
     node* leafLink;
};

struct candidate
{
    deque<int> sequence;
    int count;
};

struct inform
{
    int seqLength;
    int totalSeq;
    list<candidate>  itemSet;
};

struct returntype
{
    list<linkheader> aheader;
    node *atree;
};
```

```
returntype rtype();

typedef multimap<int, int, less<int> > sequence;
list<linkheader> lnkhdrBig;
node *rootBig;
const int an=30;
double upFreq,freqT,origFreq,oldTotalSeq,rowInDB;
double minSupp,minTol;
int runTime = 1,choose;
int maxTempPattern=0,totalBranch=0, maxL=1;;
char *origResult="total_origInfCase3Sol1S.data";
char *origResultWt="to_origInfCase3Sol1_T.data";
char *origWap="origWap.tree";
char *origLinkHeader="origLinkHeader.table";
char *SMALL="small.data";
char *sourceOrig="test1k.data";
char *sourceChanged="test2ks.data";
char *sourceUpdated="updated.data";
char *sourceInserted=sourceChanged;
char *sourceDeled=sourceChanged;
char *sourceInsertedDeled =sourceOrig;

char *total_insert_U_PL4UP="total_insert_U_PL4UP.data";
char *total_U_del_PL4UP="total_U_del_PL4UP.data";
char *result_PL4UP="result_PL4UP.data";

list<node*> item1RootSet[3000],item1RootS[3000];
list<candidate> oldSeq[an], remainedOldSeq[an],remainedNewSeq[an];
list<candidate> oldCand[an],tempSeq[an],combineSeq[an],updatedLarge[an],changedSmall[an];
list<candidate> largeToLarge[an],largeToSmall[an],largeToSmallT[an],PL,RS,switchPE;
list<inform> allInf,oldInf[an],oldI[an],remainedInf;

void BuildTree();
void BuildLinkage(node *start,list<linkheader>&lnkhdr);
positionCode* makeCode(int, positionCode*, bool);
int checkPosition(positionCode*, int, positionCode*, int);
void MiningProcess(list<node*> rootSet,list<linkheader>lnkhdr,int beginLink, queue<int> basePattern, int Count,double fre);
void miningOldTree(node* root,list<linkheader>lnkhdr,sequence seq);
void RootSetFor1Item(node* root, list<linkheader>lnkhdr,double fre, bool tT);
void compareSubSuper(list<candidate>&remainedNewS,list<candidate>remainedNewSe,short int mask[10000]);
void miningSeq(node* root, list<linkheader>lnkhdr,list<candidate>&remainedNewS,int seqL,short int mask[10000],sequence seq);
void oldSeqUpdate(node* root, list<linkheader>lnkhdr,list<candidate>& remainedS,sequence seq);

list<candidate> commonSeq(list<candidate> &oldSeq, list<candidate> &tempSeq,bool reduceCount,list<candidate> combineSeq);
list<candidate> diffSeq(list<candidate> &,list<candidate> );
list<inform> recordTotalResult(list<candidate>, int , list<inform> );
void writeTotalResult(ofstream &, list<inform>);
void readResult(ifstream & read, list<inform>& oldInf,list<candidate>& oldSeq);
void readOrigResultWtOrS(ifstream& inResult,char ts);
void writeResult(list<candidate>, ofstream&,bool ) ;
void readDatabase(ifstream & ins,double & seqNumber,sequence & duplicate,sequence & seq);
list<candidate> pruneList1(list<candidate>&,int);
list<candidate> commonPart(list<candidate> &, list<candidate> &,list<candidate> );
list<candidate> diffCommonSeq(list<candidate> &updatedL,list<candidate>&oldS,list<candidate>remainedN);
list<candidate> remainedSeq(list<candidate>&,list<candidate>,bool,list<candidate>);
sequence writeBackSeq(list<candidate> combineSeq);
void PL_RS(list<candidate> combineSeq, double origF,double minT,list<candidate>&PL,list<candidate>&RS);
node* WAPtreeCountSeq(ifstream &inFile,sequence seq,node* root,list<linkheader>&lnkhdr);
void PL4UP(int,double);
void EPL4UP(int,double);
void finalPattern(int cn, double seqNumber);
node* updatedWAPtree(ifstream& inFile1,ifstream& inF,ifstream& inFile2,sequence seqLtoS,sequence seqLtoL,sequence seqUpedL,
                     node* root1,list<linkheader>&lnkhdr1);
void Sub_update(node* tr,sequence::iterator pnt);
void Sub_combine(node* lSon, node* parent,sequence::iterator pnt);
void CheckSibling(node *tempin, node*trin);
void CheckSibling2(node *tempin, node*trin);
void CheckSibling3(node *trin, node*tempin);
void reduceCount(ifstream &inFile,node *start,sequence lToL);
```

```
void addBranch(ifstream& inFile, node* root,list<linkheader>&lnkhdr,sequence seqUpedL,bool addC, bool positionL);
void positionLinkage(node *start,list<linkheader>&lnkhdr,bool leftRight);

int main()
{
        cout << "please enter the frequency:";
        cin >> minSupp;

        minTol=0.8*minSupp;
        cout<<"minSupp="<<minSupp<<" minTol="<<minTol<<endl;

        ofstream result(result_PL4UP, ios::trunc);
        result.close();

        int tim1 = time(0);

        cout<<"Now start the program.....\n\n"<<endl;
        BuildTree();

        int tim2 = time(0);

        cout<<"\n\nbegin time: "<<tim1<<'\n';
        cout<<"end time  : "<<tim2<<'\n';
        cout<<"The execution time is: "<< tim2-tim1<< endl;
        cout<<"\n\nEnd the program"<<endl;

        return 0;
}


void BuildTree()
{
        bool OpenFile=false;

        sequence seq, duplicate,seq1,duplicate1;

        //Next is Scan the database
        int opentime=0;

        ifstream read(origResult, ios::in);

        char ts='s';
        readOrigResultWtOrS(read,ts);

        //Next is Scan the changed part
        ifstream ins (sourceChanged , ios::in);

        if ( !ins)
        {
                cerr << " File could not be opened in BuildTree(choose1):"<<opentime++<< " times\n";
                exit(1);
        }

        sequence::iterator point, eflag;
        double  seqNumber = 0;

        while (ins && !ins.eof())
        {
                readDatabase(ins,seqNumber,duplicate,seq);

        }
        upFreq=ceil(minSupp*(seqNumber+rowInDB));
        freqT = ceil(minTol*(seqNumber+rowInDB));
        origFreq=upFreq;
        cout<<"updated frequency="<<upFreq<<" .....\n";

        //record all candidate-1 items
        sequence::iterator i, bi;
        for(i = seq.begin(); i != seq.end(); i++ )
        {
```

```
            struct candidate newCand;
            newCand.sequence.push_front(i->first);
            newCand.count = i->second;
            oldCand[0].push_back(newCand);
    }
    list<candidate> oldCandTemp=oldCand[0];

    //combining old candidate-1 with changed candidate-1
    int k=0;
    cout<<"\nCombining oldCand-1 with changedCand-1.."<<endl;

    //combining the common candidate-1 and its'count
    //then add difference of combined sequences
    list<candidate> oldSeqT,oldSeqT1;
    oldSeqT=oldSeq[0];
    oldSeqT1=oldSeq[1];

    bool reduceC=false;
    combineSeq[k]=commonSeq(oldSeq[0],oldCand[0],reduceC,combineSeq[k]);
    combineSeq[k]=diffSeq(oldSeq[0],combineSeq[k]);
    combineSeq[k]=diffSeq(oldCand[0],combineSeq[k]);

    oldCand[0]=oldCandTemp;
    cout<<"\nEnd Combining Candidate-1.\n";

    combineSeq[1]=combineSeq[0];
    updatedLarge[0]=combineSeq[0];

    k=1;
    //pruning to generate combined large-1 sequence.
    combineSeq[k]=pruneList1(combineSeq[k],upFreq);
    updatedLarge[k]=combineSeq[k];

    largeToLarge[k]=commonPart(oldSeq[k],combineSeq[k],largeToLarge[k]);
    largeToSmall[k]=diffSeq(oldSeq[k],largeToSmall[k]);
    switchPE=diffSeq(combineSeq[k],switchPE);

    double origF=ceil(minSupp*rowInDB);
    double minT=ceil(minTol*rowInDB);
    oldSeq[0]=oldSeqT;
    oldSeq[1]=oldSeqT1;
    combineSeq[1]=updatedLarge[1];
    PL_RS(oldSeqT, origF, minT,PL,RS);
    list<candidate> PLtemp, RStemp;
    PLtemp=PL;
    RStemp=RS;

    if(switchPE.size()==0)
    {
            cout<<"\ncase1 or 2"<<endl;
            int cn=12;
            PL4UP(cn,seqNumber);
    }
    else
    {
            ifstream inResult(origResultWt,ios::in);
            list<candidate> switchC, switchCa,switchCas,switchCase;
            switchC=commonPart(switchPE,PL,switchC);
            switchCa=diffSeq(switchPE,switchCa);
            PL=PLtemp;
            if(switchCa.size()==0)
            {
                    cout<<"\ncase3 solution1"<<endl;
                    int cn=31;//case3 solution1
                    char ts='t';
                    readOrigResultWtOrS(inResult,ts);
                    PL4UP(cn,seqNumber);
            }
            else
            {
```

```
                int sizeRS=RS.size();
                cout<<"sizeRS:"<<sizeRS<<"  ";
                switchCas=commonPart(switchCa,RS,switchCas);
                switchCase=diffSeq(switchCa,switchCase);
                int sizeR=RS.size();
                RS=RStemp;
                cout<<"sizeR:"<<sizeR<<"  ";
                if(switchCase.size()==0)
                {
                        cout<<"\ncase3 solution2"<<endl;
                        int cn=32;//case3 solution2
                        EPL4UP(cn,seqNumber);
                }
                else if(sizeRS != sizeR)
                {
                        cout<<"\ncase5 solution2"<<endl;
                        int cn=52;//case5 solution2
                        EPL4UP(cn,seqNumber);
                }
                else
                {
                        cout<<"\ncase5 solution1"<<endl;
                        int cn=51;//case5 solution1
                        char ts='t';
                        readOrigResultWtOrS(inResult,ts);
                        PL4UP(cn,seqNumber);
                }
        }
    }


    cout<<"Finish mining process,max length of seq: "<<maxTempPattern<<"-sequence..."<<endl;
    ofstream result2;

    allInf.clear();
    result2.open(total_insert_U_PL4UP, ios::trunc);
    int longSeq=max(maxTempPattern+1,oldTotalSeq);
    cout<<" maxTemPattern "<<maxTempPattern<<" oldTotalSeq "<<oldTotalSeq;
    result2<<longSeq<<"\t"<<seqNumber+rowInDB<<endl;
    for(int h=0; h<=longSeq; h++)
    {
            allInf=recordTotalResult(updatedLarge[h],h,allInf);
    }
    writeTotalResult(result2,allInf);
    cout<<"\n";
    return;

}//end BuildTree


void PL4UP(int cn, double seqNumber)
{
    //retrieve the tree and link header table
    returntype returned=rtype();
    list<linkheader> lnkhdr1=returned.aheader;
    node *root1=returned.atree;

    sequence seqOldL = writeBackSeq(oldSeq[1]);
    linkheader *newLinkHeader;
    lnkhdr1.clear();
    for (sequence::iterator  i = seqOldL.begin(); i!=seqOldL.end(); i++)
    {
            newLinkHeader = new linkheader;
            newLinkHeader->link = NULL;
            newLinkHeader->lastLink = NULL;
            newLinkHeader->event= i->first;
            newLinkHeader->occur= i->second;
            lnkhdr1.push_back(*newLinkHeader);
            free(newLinkHeader);
    }
```

```
cout<<"End of building tree and begin to build linkage..."<<endl;
positionLinkage(root1->lSon,lnkhdr1,true);
cout<<"End of building linkage...\n";

rootBig=root1;
lnkhdrBig=lnkhdr1;

list<candidate> switchC, switchCa, switchCas, switchCase;

if(cn==12)//if-cn12
{
        switchC=largeToSmall[1];
        if(switchC.size()==0)//if-switchC
        {
                int case1=1;
                cout<<"\ncase 1 begin..\n";
                finalPattern(case1,seqNumber);
        }//end if-switchC
        else
        {
                int case2=2;
                cout<<"\ncase 2 begin..\n";
                finalPattern(case2,seqNumber);
        }
}//end if-cn12

if(cn==31 || cn==51)//if-cn3151
{
        switchC=oldSeq[1];
        switchC=diffSeq(PL,switchC);
        switchCa=combineSeq[1];
        switchCas=commonPart(switchC,switchCa,switchCas);
        switchCase=diffSeq(switchC,switchCase);
        largeToSmallT[1]=switchCase;

        if(switchCase.size()==0)
        {
                int case3511=3511;
                cout<<"\ncase 3511 begin..\n";
                finalPattern(case3511,seqNumber);
        }
        else
        {
                int case3512=3512;
                cout<<"\ncase 3512 begin..\n";
                finalPattern(case3512,seqNumber);
        }
}//end if-cn3151


}//end PL4UP

void EPL4UP(int cn, double seqNumber)
{
        //retrieve the tree and link header table
        returntype returned=rtype();
        list<linkheader> lnkhdr1=returned.aheader;
        node *root1=returned.atree;

        node *root2;
        list<linkheader> lnkhdr2;
        ifstream inFile1(SMALL);
        ifstream inF(SMALL);
        ifstream inFile2(sourceChanged);
        sequence seqLargeToS = writeBackSeq(largeToSmall[1]);
        sequence seqLargeToL = writeBackSeq(largeToLarge[1]);
        sequence seqUpedL = writeBackSeq(updatedLarge[1]);

        root2=updatedWAPtree(inFile1,inF,inFile2,seqLargeToS,seqLargeToL,seqUpedL,root1,lnkhdr1);
```

```
        rootBig=root2;
        lnkhdrBig=lnkhdr1;
        if(cn==32)
                cout<<"\ncase 32 begin...\n";
        if(cn==52)
                cout<<"\ncase 52 begin...\n";

        finalPattern(cn, seqNumber);

}


void finalPattern(int cn, double seqNumber)
{

        double insertF=ceil(minSupp*seqNumber);
        double insertT=ceil(minTol*seqNumber);

        if(largeToSmall[1].size()!=0)
        {
                for(int ii=2; ii<=oldTotalSeq; ii++)
                {
                        remainedOldSeq[ii]=remainedSeq(oldSeq[ii],largeToSmall[1],false,remainedOldSeq[ii]);
                }
        }
        else
        {
                for(int ii=2; ii<=oldTotalSeq; ii++)
                {
                        remainedOldSeq[ii]=oldSeq[ii];
                }
        }

        list<candidate> tempCombineSeq=combineSeq[1];
        oldCand[1]=commonPart(tempCombineSeq,oldCand[0],oldCand[1]);
        list<candidate> ::iterator back=oldCand[1].begin();
        seq.clear();
        for(; back!=oldCand[1].end(); back++ )
        {
                //struct candidate newCand;
                deque<int> newCandS=back->sequence;
                int newCandC =back->count;
                seq.insert(sequence::value_type(newCandS[0],newCandC));
        }

        cout<<"Finish scanning the insert db and Begin to build the tree ... " << endl;

        // Next is build the small WAP tree
        ifstream inFile;
        inFile.open(sourceInserted , ios::in);

        node *root = new node;
        root->event = -1;
        root->occur = seqNumber;//+rowInDB
        root->CountSon =0;
        root->pcLength =0;
        root->parent = NULL;
         root->lSon = NULL;
        root->rSibling = NULL;
        root->nextLink = NULL;

        list<linkheader> lnkhdr;


        root=WAPtreeCountSeq(inFile,seq,root,lnkhdr);

        cout<<"End of building tree and linkage..."<<endl;
        cout<<" L1 total seq:"<<seq.size()<<endl;
```

```
//begin mining process
if(seq.size()==0)
        cout<<"Cannot find 1-sequence in changed part,please lower min sup.\n";
else//else 1 begin.
{
        cout<<"Found 1-sequence, begin to update oldSeq oldTotalSeq="<<oldTotalSeq<<endl;
        int remainedL=1;
        if(cn==32 || cn==52)
                RootSetFor1Item(root,lnkhdr,insertF,true);
        else
                RootSetFor1Item(root,lnkhdr,insertT,true);

        for(int ii=2; ii<=oldTotalSeq; ii++)
        {
                oldSeqUpdate(root, lnkhdr, oldSeq[ii],seq);
                if(oldSeq[ii].size()!=0)
                remainedL++;
        }
        oldTotalSeq=remainedL;
        cout<<"finish oldSeqUpdate, oldTotalSeq="<<oldTotalSeq;

        list<node*> newRootSet;
        queue<int> beginPattern;
        int controlBrow=0;

        newRootSet.push_back(root);
        cout<<"\nBegin the mining process"<<endl;

        //mining 2nd part of Patterns and put in updatedLarge[ii]
        if(cn==1 || cn==2 || cn==32 || cn==52)
        {
                MiningProcess(newRootSet,lnkhdr,controlBrow, beginPattern, seqNumber,insertF);//+
        }
        else if(cn==3511 || cn==3512)
        {
                MiningProcess(newRootSet,lnkhdr,controlBrow, beginPattern, seqNumber,insertT);//+
        }
        cout<<"Finish mining process,max length of seq: "<<maxTempPattern<<"-sequence..."<<endl;

        for(ii=2; ii<=maxTempPattern;ii++)//+1
        {
                remainedNewSeq[ii]=diffCommonSeq(updatedLarge[ii],oldSeq[ii],remainedNewSeq[ii]);
        }

}//else 1 end


int maxOld=1;
int maxC=maxTempPattern;

if(remainedNewSeq[2].size() !=0)
{
        cout<<" begin for mining big tree: ";
        //count supports in oldTree then prune candidates in changed part
        sequence seqOldL = writeBackSeq(oldSeq[1]);
        sequence seqUpdatedL=writeBackSeq(updatedLarge[1]);
        if(cn==32 || cn==52)
                miningOldTree(rootBig, lnkhdrBig,seqUpdatedL);//update 2nd part
        else
                miningOldTree(rootBig, lnkhdrBig,seqOldL);//update 2nd part
        int seqlength=max(maxTempPattern,oldTotalSeq);

        for(int ii=2; ii<=seqlength;ii++)
        {
                updatedLarge[ii]=diffSeq(updatedLarge[ii],oldSeq[ii]);//largeToLarge
                if(updatedLarge[ii].size()!=0)
                        maxOld++;
        }
}
else
```

```
        {
                cout<<" remainedNewSeq[2].size()="<<remainedNewSeq[2].size();
                for(int ii=2; ii<=oldTotalSeq;ii++)
                {
                        updatedLarge[ii]=oldSeq[ii];
                        if(updatedLarge[ii].size()!=0)
                                maxOld++;
                }
        }

        oldTotalSeq=maxOld;

}

//=====================below are functions==========

node* WAPtreeCountSeq(ifstream &inFile,sequence seq,node* root,list<linkheader>&lnkhdr)
{
        linkheader *newLinkHeader;
        node *Tranversal, *newNode, *Parent;
        int cid,number,event;

        if ( !inFile)
        {
                cerr << " File could not be opened\n";
                exit(1);
        }

        while (inFile && !inFile.eof())//while-inFile
        {
                deque<int>inSequence;

                inFile >> cid;
                inFile >> number;

                Tranversal = root;

                for(int i=0; i< number; i++)//for-int
                {
                        inFile >> event;

                        inSequence.push_back(event);

                        if(seq.find(event) != seq.end())//if-seq
                        {
                                Parent = Tranversal;

                                if( Tranversal->lSon == NULL)
                                {
                                        newNode = new node;
                                        newNode->event = event;
                                        newNode->occur = 1;
                                        newNode->lSon = NULL;
                                        newNode->rSibling = NULL;
                                        newNode->nextLink = NULL;
                                        newNode->CountSon = 0;
                                        Parent->CountSon ++;
                                        newNode->parent = Parent;
                                        newNode->pcLength = Tranversal->pcLength + 1;
                                        newNode->pcCode = makeCode(Tranversal->pcLength,Tranversal->pcCode,true);
                                        Tranversal->lSon = newNode;
                                        Tranversal = newNode;
                                }
                                else
                                {
                                        Tranversal = Tranversal->lSon;
                                        if ( Tranversal->event == event)
                                        {
                                                (Tranversal->parent)->CountSon++;
                                                Tranversal->occur++;
```

```
                                        }

                                        else
                                        {
                                                bool find= false;
                                                while(Tranversal->rSibling != NULL && !find )
                                                {
                                                        Tranversal = Tranversal->rSibling;

                                                        if ( Tranversal->event == event)
                                                        {
                                                                Tranversal->occur ++;
                                                                (Tranversal->parent)->CountSon++;
                                                                find = true;
                                                        }
                                                }
                                                if (!find)
                                                {
                                                        newNode = new node;
                                                        newNode->event = event;
                                                        newNode->occur = 1;
                                                        newNode->lSon = NULL;
                                                        newNode->rSibling = NULL;

                                                        newNode->nextLink = NULL;
                                                        newNode->CountSon = 0;
                                                        Parent->CountSon ++;
                                                        newNode->parent = Parent;
                                                        newNode->pcLength = Tranversal->pcLength + 1;
                                                newNode->pcCode = makeCode(Tranversal->pcLength,Tranversal->pcCode, false);

                                                        Tranversal->rSibling = newNode;
                                                        Tranversal = newNode;

                                                }
                                        }
                                }
                        } //end if-seq
                }//end for-int
        }//end while-inFile

        // next tranversing all tree to build the Pre-order linkage
        for (sequence::iterator  i = seq.begin(); i!=seq.end(); i++)
        {
                newLinkHeader = new linkheader;
                newLinkHeader->link = NULL;
                newLinkHeader->lastLink = NULL;
                newLinkHeader->event= i->first;
                newLinkHeader->occur= i->second;
                lnkhdr.push_back(*newLinkHeader);
                free(newLinkHeader);
        }
        cout<<"End of building tree and begin to build linkage..."<<endl;
        BuildLinkage(root->lSon,lnkhdr);
        cout<<"End of building linkage...\n";

        return root;
}//end WAPtreeCountSeq

void MiningProcess(list<node*> rootSet,list<linkheader>lnkhdr,int beginLink, queue<int> basePattern, int Count,double fre)
{
        list<linkheader>::iterator pnt;
        pnt = lnkhdr.begin();
        runTime++;
        for( ; pnt != lnkhdr.end(); pnt++)
        {
                list<node*> newRootSet;
                node * SavePoint = NULL;
                bool DescSave = false;
                bool RootUsed = false;
```

```
int totalSon = 0;
int count = 0;
int emptySon = 0;
int RootCount = Count;

node * linkBrow = pnt -> link;
list<node*>::iterator rootBrow = rootSet.begin();
while (linkBrow != NULL && rootBrow != rootSet.end() && RootCount >= fre )
 {
        int check = checkPosition(((*rootBrow)->lSon)->pcCode, (*rootBrow)->pcLength+1,
                                  linkBrow->pcCode, linkBrow->pcLength);
        switch ( check )
        {
                case 0:
                        if ( SavePoint != NULL )
                        {
                                if ( SavePoint->lSon == NULL )
                                DescSave = false;
                                else
                                if (checkPosition( (SavePoint->lSon)->pcCode, SavePoint->pcLength+1,
                                        linkBrow->pcCode, linkBrow->pcLength)== 0)
                                DescSave = true;
                                else DescSave = false;
                        }

                        if ( !DescSave)
                        {
                                count = count + linkBrow->occur;
                                totalSon = totalSon + linkBrow->CountSon;
                                RootUsed = true;
                                SavePoint = linkBrow;
                                if (linkBrow->lSon != NULL)
                                        newRootSet.push_back(linkBrow);
                                else
                                        emptySon = emptySon + linkBrow->occur;
                        }
                        linkBrow = linkBrow->nextLink;
                        break;
                case 1:
                        if (!RootUsed)
                        RootCount = RootCount - (*rootBrow)->occur;
                        rootBrow++;
                        RootUsed = false;
                        break;
                case 2:
                        linkBrow = linkBrow->nextLink;
                        break;
                case 3:
                        linkBrow = linkBrow->nextLink;
                        break;
        }//end switch
}//end while

if ( count >= fre)//if-count
{
        //record all length seq
        queue<int> temp=basePattern;
        deque<int> convertPattern;
        temp.push(pnt->event);

        while(!temp.empty())
        {
                convertPattern.push_back(temp.front());
                temp.pop();
        }

        struct candidate newCand;
        newCand.sequence=convertPattern;
        newCand.count=count;
```

```
                                if(convertPattern.size()>1)
                                        updatedLarge[convertPattern.size()].push_back(newCand);

                                queue<int> tempPattern = basePattern;
                                tempPattern.push(pnt->event);
                                queue<int> otherPattern = tempPattern;

                                if(maxTempPattern<tempPattern.size())
                                        maxTempPattern=tempPattern.size();

                                int beginL=0;

                                ofstream result(result_PL4UP, ios::app);
                                while(!tempPattern.empty())
                                {
                                 result<<tempPattern.front()<<"\t";
                                tempPattern.pop();
                                }
                                result<<endl;

                                if ( totalSon >=fre)
                                {
                                        MiningProcess(newRootSet, lnkhdr,beginL,otherPattern, count-emptySon,fre);
                                }
                        }//end if-count
                }//end for

        return;
}//end MiningProcess

void miningOldTree(node* root, list<linkheader>lnkhdr,sequence seq)
{
        cout<<"enter RootSetFor1Item\n";
        double fre=ceil(minSupp*rowInDB);
        RootSetFor1Item(root, lnkhdr,fre,false);
        //accumulating support of candidates of remainedNewSeq[ii] in changed part or updated part
        int subL=0,remainedL=1,checkSeqL=maxTempPattern-1;

        for(int ii=maxTempPattern; ii>=2;ii--)
        {
                short int mask[10000]={0};
                if(remainedNewSeq[ii].size()!=0)
                {
                        subL++;
                        if(subL==1)
                        {
                                miningSeq( root, lnkhdr,remainedNewSeq[ii],ii,mask,seq);
                                updatedLarge[ii]=remainedNewSeq[ii];
                                if(remainedNewSeq[ii].size()!=0)
                                        remainedL++;
                        }
                        else
                        {
                                compareSubSuper(remainedNewSeq[ii],remainedNewSeq[ii+1].mask);
                                miningSeq(root, lnkhdr,remainedNewSeq[ii],ii,mask,seq);
                                updatedLarge[ii]=remainedNewSeq[ii];//diffSeq(remainedNewSeq[ii],updatedLarge[ii]);
                                if(remainedNewSeq[ii].size()!=0)
                                        remainedL++;
                        }
                }
        }//end for
        maxTempPattern=remainedL;
}//end miningOldTree


void compareSubSuper(list<candidate>&updatedL,list<candidate>oldS,short int mask[10000])
{
        list<candidate>::iterator firstBrow,firstTemp, secondBrow, secondTemp;
        int m=0;
        firstBrow=updatedL.begin();
```

```
		secondBrow=oldS.begin();
		bool compareF = false;
		struct candidate newCand;

		while(compareF != true)//outer-while
		{
			bool matchFlag=false,move1=false,move2=false,record=false;
			int j1=0, k2=0;

			deque<int> firstSeq=firstBrow->sequence;
			int firstCount=firstBrow->count;
			deque<int> secondSeq=secondBrow->sequence;
			int secondCount=secondBrow->count;

			while(j1<firstSeq.size()&& k2<secondSeq.size()&&!matchFlag&&!move1&&!move2)
			{//inner-while
				if(firstSeq[j1]==secondSeq[k2])
				{
					j1++;
					if(j1==firstSeq.size())
						matchFlag=true;
				}
					k2++;


			}//inner-while

			if(matchFlag==true)
			{
				firstBrow->count=secondBrow->count;
				mask[m]=1;
				if(firstBrow!=updatedL.end())
				{
					firstBrow++;
					secondBrow=oldS.begin();
					m++;
				}
				else
					compareF=true;
			}
			else
			{
				if(secondBrow!=oldS.end())
					secondBrow++;
				else
				{
					if(firstBrow!=updatedL.end())
					{
						firstBrow++;
						m++;
						secondBrow=oldS.begin();
					}
					else
						compareF=true;
				}
			}
		}//end outer-while
		return;

}//end compareSubSuper

void RootSetFor1Item(node* root, list<linkheader>lnkhdr,double frequency,bool tOrT)
{
		list<node*> rootSet;
		rootSet.push_back(root);
		int Count=root->CountSon;

		int m=0;
		list<linkheader>::iterator pnt;
```

```
for( pnt = lnkhdr.begin(); pnt != lnkhdr.end(); pnt++)
{
    list<node*> newRootSet;
    node * SavePoint = NULL;
    bool DescSave = false;
    bool RootUsed = false;
    int totalSon = 0;
    int count = 0;
    int emptySon = 0;
    int RootCount = Count;

    node * linkBrow = pnt -> link;
    list<node*>::iterator rootBrow = rootSet.begin();

    while (linkBrow != NULL && rootBrow != rootSet.end() && RootCount >= frequency )
    {

        int check = checkPosition(((*rootBrow)->lSon)->pcCode, (*rootBrow)->pcLength+1,
                                  linkBrow->pcCode, linkBrow->pcLength);
        switch ( check )
        {
            case 0:
                if ( SavePoint != NULL )
                {
                        if ( SavePoint->lSon == NULL )
                        DescSave = false;
                        else
                        if (checkPosition( (SavePoint->lSon)->pcCode, SavePoint->pcLength+1,
                                linkBrow->pcCode, linkBrow->pcLength)== 0)
                        DescSave = true;
                        else DescSave = false;
                }

                if ( !DescSave)
                {
                        totalSon = totalSon + linkBrow->CountSon;
                        RootUsed = true;
                        SavePoint = linkBrow;
                        if (linkBrow->lSon != NULL)
                                newRootSet.push_back(linkBrow);
                        else
                                emptySon = emptySon + linkBrow->occur;
                }
                linkBrow = linkBrow->nextLink;
                break;

            case 1:
                if (!RootUsed)
                RootCount = RootCount - (*rootBrow)->occur;
                rootBrow++;
                RootUsed = false;
                break;
            case 2:
                linkBrow = linkBrow->nextLink;
                break;
            case 3:
                linkBrow = linkBrow->nextLink;
                break;
        }//end swithch

    }// end while

    if(tOrT==true)
        itemIRootS[m]=newRootSet;
    else
        itemIRootSet[m]=newRootSet;
        m++;
}//end for
return;
```

```
}//end RootSetForIItem

void miningSeq(node* root, list<linkheader>lnkhdr,list<candidate>&remainedNewS,int seqL,short int mask[10000],sequence seq)
{
        int m=0;
        list<candidate>::iterator candBrow = remainedNewS.begin();
        while(candBrow != remainedNewS.end())
        {//for-list
                if(mask[m]==0)
                {
                        deque<int> candSequence = candBrow->sequence;
                        int seqC=candBrow->count;
                        bool inOldTree=true;
                        bool flag = false;

                        for(int jj=0; jj<candSequence.size(); jj++)
                        {
                                if(seq.find(candSequence[jj])==seq.end())
                                        inOldTree=false;
                        }
                        unsigned int j = 1, k = 0;
                        list<node*> rootSet;
                        int Count=root->CountSon;
                        int newCount;

                        list<linkheader>::iterator pnt;
                        pnt=lnkhdr.begin();

                        if(inOldTree)
                        {
                                while((*pnt).event!=candSequence[0])
                                {
                                        pnt++;
                                        k++;
                                }
                        }

                        rootSet=itemIRootSet[k];

                        while(j<candSequence.size()&& !flag && inOldTree)
                        {//while-j
                                list<linkheader>::iterator pntI;
                                pntI=lnkhdr.begin();
                                while((*pntI).event!=candSequence[j])
                                        pntI++;

                                list<node*> newRootSet;
                                node * SavePoint = NULL;
                                bool DescSave = false;
                                bool RootUsed = false;
                                int totalSon = 0;
                                int count = 0;
                                int emptySon = 0;
                                int RootCount = Count;

                                node * linkBrow = pntI -> link;

                                list<node*>::iterator rootBrow = rootSet.begin();

                                while (linkBrow != NULL && rootBrow != rootSet.end() )
                                {//while-linkBrow
                                        int check = checkPosition(((*rootBrow)->lSon)->pcCode, (*rootBrow)->pcLength+1,
                                                                linkBrow->pcCode, linkBrow->pcLength);

                                        switch ( check )
                                        {
                                                case 0:
                                                        if ( SavePoint != NULL )
                                                        {
```

```
                                        if ( SavePoint->lSon == NULL )
                                                DescSave = false;
                                        else
                                            if (checkPosition( (SavePoint->lSon)->pcCode,
                                            SavePoint->pcLength+1, linkBrow->pcCode, linkBrow->pcLength)== 0)
                                                    DescSave = true;
                                             else DescSave = false;
                                }

                                if ( !DescSave)
                                {
                                        count = count + linkBrow->occur;
                                        totalSon = totalSon + linkBrow->CountSon;
                                        RootUsed = true;
                                        SavePoint = linkBrow;
                                        if (linkBrow->lSon != NULL)
                                                newRootSet.push_back(linkBrow);
                                        else
                                                emptySon = emptySon + linkBrow->occur;
                                }

                                linkBrow = linkBrow->nextLink;
                                break;
                        case 1:
                                if (!RootUsed)
                                RootCount = RootCount - (*rootBrow)->occur;
                                rootBrow++;
                                RootUsed = false;
                                break;
                         case 2:
                                linkBrow = linkBrow->nextLink;
                                break;
                        case 3:
                                linkBrow = linkBrow->nextLink;
                                break;
                        }//end switch
                }//end while-linkBrow

                if ( (count+seqC) < upFreq)//if-count
                {
                        flag=true;
                }
                else
                {
                        rootSet=newRootSet;
                        Count=count-emptySon;
                        j++;
                        newCount=count+seqC;
                }

}//end while-j

if(inOldTree)
{
        if(flag==true )
        {
                list<candidate>::iterator tempBrow=candBrow;
                candBrow++;
                remainedNewS.erase(tempBrow);
        }
        else
        {
                candBrow->count=newCount;
                candBrow++;
        }
}
else
{
```

```
                                        if(seqC<upFreq)
                                        {
                                                list<candidate>::iterator tempBrow=candBrow;
                                                candBrow++;
                                                remainedNewS.erase(tempBrow);
                                        }
                                        else
                                        {
                                                candBrow++;
                                        }
                                }
                                m++;

                        }//end if-mask
                        else
                        {
                                candBrow++;
                                m++;
                        }

                }//end for-list

        return;
}//end miningSeq

void oldSeqUpdate(node* root, list<linkheader>lnkhdr,list<candidate>& remainedNewS,sequence seq)
{
        list<candidate>::iterator candBrow = remainedNewS.begin();
        while(candBrow != remainedNewS.end())
        {//for-list
                deque<int> candSequence = candBrow->sequence;
                int seqC=candBrow->count;
                bool inOldTree=true;
                bool flag = false;

                for(int jj=0; jj<candSequence.size(); jj++)
                {
                        if(seq.find(candSequence[jj])==seq.end())
                                inOldTree=false;
                }
                unsigned int j = 1, k = 0;
                list<node*> rootSet;
                int Count=root->CountSon;
                int newCount;

                list<linkheader>::iterator pnt;
                pnt=lnkhdr.begin();
                if(inOldTree)
                {
                        while((*pnt).event!=candSequence[0])
                        {
                                pnt++;
                                k++;
                        }
                }

                rootSet=item1RootS[k];

                while(j<candSequence.size()&& !flag && inOldTree)
                {//while-j
                        list<linkheader>::iterator pnt1;
                        pnt1=lnkhdr.begin();
                        while((*pnt1).event!=candSequence[j])
                                pnt1++;

                        list<node*> newRootSet;
                        node * SavePoint = NULL;
                        bool DescSave = false;
                        bool RootUsed = false;
                        int totalSon = 0;
```

```
int count = 0;
int emptySon = 0;
int RootCount = Count;

node * linkBrow = pnt1 -> link;

list<node*>::iterator rootBrow = rootSet.begin();

while (linkBrow != NULL && rootBrow != rootSet.end() )
{//while-linkBrow
        int check = checkPosition(((*rootBrow)->lSon)->pcCode, (*rootBrow)->pcLength+1,
                                  linkBrow->pcCode, linkBrow->pcLength);

        switch ( check )
        {
            case 0:
                if ( SavePoint != NULL )
                {
                        if ( SavePoint->lSon == NULL )
                                DescSave = false;
                        else
                        if (checkPosition( (SavePoint->lSon)->pcCode, SavePoint->pcLength+1,
                                linkBrow->pcCode, linkBrow->pcLength)== 0)
                                DescSave = true;
                        else DescSave = false;
                }

                if ( !DescSave)
                {
                        count = count + linkBrow->occur;
                        totalSon = totalSon + linkBrow->CountSon;

                        RootUsed = true;
                        SavePoint = linkBrow;
                        if (linkBrow->lSon != NULL)
                                newRootSet.push_back(linkBrow);
                        else
                                emptySon = emptySon + linkBrow->occur;
                }

                linkBrow = linkBrow->nextLink;
                 break;
            case 1:
                if (!RootUsed)
                        RootCount = RootCount - (*rootBrow)->occur;
                rootBrow++;
                RootUsed = false;
                break;
            case 2:
                linkBrow = linkBrow->nextLink;
                break;
            case 3:
                linkBrow = linkBrow->nextLink;
                break;
        }//end switch
}//end while-linkBrow

if ( (count+seqC) < upFreq)//if-count
{
        flag=true;
}
else
{
        rootSet=newRootSet;
        Count=count-emptySon;
        j++;

        newCount=count+seqC;
}
```

```
                    }//end while-j


                    if(inOldTree)
                    {
                            if(flag==true )
                            {
                                    list<candidate>::iterator tempBrow=candBrow;
                                    candBrow++;
                                    remainedNewS.erase(tempBrow);
                            }
                            else
                            {
                                    candBrow->count=newCount;
                                    candBrow++;
                            }
                    }
                    else
                    {
                            if(seqC<upFreq)
                            {
                                    list<candidate>::iterator tempBrow=candBrow;
                                    candBrow++;
                                    remainedNewS.erase(tempBrow);
                            }
                            else
                            {
                                    candBrow++;
                            }
                    }
            }//end for-list

    return;

}//end oldSeqUpdate

void BuildLinkage(node *start,list<linkheader>&lnkhdr)
{
    if (start !=NULL)
    {
        list<linkheader>::iterator lnkBrow = lnkhdr.begin();
        while (lnkBrow->event != start->event && lnkBrow != lnkhdr.end())
            lnkBrow++;

        node *lastLinkage;
        lastLinkage = lnkBrow->lastLink;
        if (lastLinkage == NULL )
            lnkBrow->link = start;
        else
            lastLinkage->nextLink = start;
        lnkBrow->lastLink = start;

        BuildLinkage(start->lSon,lnkhdr);
        BuildLinkage(start->rSibling,lnkhdr);
    }
    else return;
}

//pruning list seq1
list<candidate> pruneList1(list<candidate>&combineSeq,int upF)
{
                    list<candidate>::iterator ui, bui;
                    ui=bui=combineSeq.begin();
                    while(ui != combineSeq.end())
                    {
                            if(ui->count < upF)
                            {
                                    bui++; combineSeq.erase(ui); ui=bui;
                            }
                            else
```

```
                        {
                                if(bui != combineSeq.end())
                                {
                                        ui++; bui++;
                                }
                                else
                                        ui=bui;
                        }
                }
                return combineSeq;
}//pruneList1

//extracting common part of 2 sequence and add two counts into common
list<candidate> commonSeq(list<candidate> &oldSeq, list<candidate> &tempSeq,bool reduceCount,list<candidate> combineSeq)
{
        list<candidate>::iterator itemBrow,oldTemp, changedBrow, changedTemp;
        itemBrow=oldSeq.begin();
        changedBrow=tempSeq.begin();
        bool compareF = false;
        struct candidate newCand;

        while( compareF != true)//common
        {
                bool matchFlag=false;
                int jj=0, kk=0;

                deque<int> infSeq=itemBrow->sequence;
                int infCount=itemBrow->count;
                deque<int> changedSeq=changedBrow->sequence;
                int changedCount=changedBrow->count;

                while(jj<infSeq.size()&& kk<changedSeq.size()&&!matchFlag)
                {
                        if(infSeq[jj]==changedSeq[kk])
                        {
                                jj++;
                                if(jj==infSeq.size())
                                        matchFlag=true;
                        }
                        kk++;
                }

                if(matchFlag==true)
                {
                        //cout<<"if";
                        newCand.sequence=infSeq;
                        if(reduceCount==true)
                        {
                                newCand.count = infCount-changedCount;
                        }
                        else
                        {
                                newCand.count = infCount+changedCount;
                        }
                        combineSeq.push_back(newCand);

                        oldTemp=itemBrow;
                        changedTemp=changedBrow;
                        itemBrow++;
                        changedBrow++;
                        oldSeq.erase(oldTemp);
                        tempSeq.erase(changedTemp);

                        changedBrow=tempSeq.begin();

                        matchFlag=false;
                }
                else
                {       if (changedBrow != tempSeq.end())
                                changedBrow++;
```

```
                                else if(itemBrow != oldSeq.end())
                                {
                                        itemBrow++;
                                        changedBrow=tempSeq.begin();
                                }
                                        else
                                        compareF=true;
                        }
                }//end while common
                        return combineSeq;
}
//end of commonSeq

//appending differece part of 2 sequence
list<candidate> diffSeq(list<candidate> &tempSeq,list<candidate> combineSeq)
{
        struct candidate newCand;
        list<candidate>::iterator itemBrow, changedBrow;

        for(changedBrow=tempSeq.begin(); changedBrow != tempSeq.end(); changedBrow++)
        {
                deque<int> changedSeq=changedBrow->sequence;
                int changedCount=changedBrow->count;
                newCand.sequence=changedSeq;
                newCand.count = changedCount;
                combineSeq.push_back(newCand);
        }
        return combineSeq;
}//end of diffSeq

//obtain common part from 2 seq, but not add count
list<candidate> commonPart(list<candidate>& oldSeq, list<candidate>& tempSeq,list<candidate> combineSeq)
{
        list<candidate>::iterator itemBrow,oldTemp, changedBrow, changedTemp;
        itemBrow=oldSeq.begin();
        changedBrow=tempSeq.begin();
        bool compareF = false;
        struct candidate newCand;

        while(itemBrow != oldSeq.end() && compareF != true)//common
        {
                bool matchFlag=false;
                int jj=0, kk=0;

                deque<int> infSeq=itemBrow->sequence;
                int infCount=itemBrow->count;
                deque<int> changedSeq=changedBrow->sequence;
                int changedCount=changedBrow->count;

                while(jj<infSeq.size()&& kk<changedSeq.size()&&!matchFlag)
                {
                        if(infSeq[jj]==changedSeq[kk])
                        {
                                jj++;
                                if(jj==infSeq.size())
                                        matchFlag=true;
                        }
                        kk++;
                }

                if(matchFlag==true)
                {
                        newCand.sequence=infSeq;
                        newCand.count = changedCount;
                        combineSeq.push_back(newCand);

                        oldTemp=itemBrow;
                        changedTemp=changedBrow;
                        itemBrow++;
                        changedBrow++;
```

```
                                oldSeq.erase(oldTemp);
                                tempSeq.erase(changedTemp);

                                changedBrow=tempSeq.begin();
                        }
                        else if (changedBrow != tempSeq.end())
                                        changedBrow++;
                                else if(itemBrow != oldSeq.end())
                                {
                                        itemBrow++;
                                        changedBrow=tempSeq.begin();
                                }
                                        else
                                        compareF=true;
        }//end while common
                        return combineSeq;
}
//end of commonPart


list<candidate> diffCommonSeq(list<candidate> &updatedL,list<candidate>&oldS,list<candidate>remainedN)
{

        list<candidate>::iterator firstBrow,firstTemp, secondBrow, secondTemp;

        firstBrow=updatedL.begin();
        secondBrow=oldS.begin();
        bool compareF = false;
        struct candidate newCand;

        while(compareF != true)//outer-while
        {
                bool matchFlag=false,move1=false,move2=false,record=false;
                int j1=0, k2=0;

                deque<int> firstSeq=firstBrow->sequence;
                int firstCount=firstBrow->count;
                deque<int> secondSeq=secondBrow->sequence;
                int secondCount=secondBrow->count;

                while(j1<firstSeq.size()&& k2<secondSeq.size()&&!matchFlag&&!move1&&!move2)
                {//inner-while
                        if(firstSeq[j1]==secondSeq[k2])
                        {
                                j1++;
                                if(j1==firstSeq.size())
                                        matchFlag=true;

                                k2++;
                        }
                        else if(firstSeq[j1]<secondSeq[k2])
                        {
                                record=true;
                                move1=true;
                        }
                        else if(firstSeq[j1]>secondSeq[k2])
                        {
                                move2=true;
                        }

                }//inner-while

                if(record==true)
                {
                        newCand.sequence=firstSeq;
                        newCand.count = firstCount;
                        remainedN.push_back(newCand);
                }

                if(matchFlag==true)
```

```
                {
                        list<candidate>::iterator temp;
                        temp=firstBrow;
                        firstBrow++;
                        updatedL.erase(temp);

                        secondBrow++;
                }

                if(move1==true)
                {
                        firstBrow++;
                }
                if(move2==true)
                {
                        secondBrow++;
                }

                if (firstBrow == updatedL.end())
                        compareF=true;
                else if(secondBrow == oldS.end())
                {
                        while(firstBrow != updatedL.end())
                        {
                                firstSeq=firstBrow->sequence;
                                firstCount=firstBrow->count;

                                newCand.sequence=firstSeq;
                                newCand.count = firstCount;
                                remainedN.push_back(newCand);
                                firstBrow++;
                        }
                        compareF=true;
                }

        }//end outer-while
                return remainedN;
}//end diffCommonSeq


//using largeToSmall-1 to check all oldSeq and erase them
list<candidate> remainedSeq(list<candidate> &oldSeq,list<candidate>largeToSmall,bool erase2Seq,list<candidate>remainedOldSeq)
{
        list<candidate>::iterator itemBrow,oldTemp, changedBrow, changedTemp;
        itemBrow=oldSeq.begin();
        changedBrow=largeToSmall.begin();
        bool compareF = false;
        struct candidate newCand;
        while( compareF != true)//common
        {
                bool matchFlag=false;
                int jj=0, kk=0;

                deque<int> infSeq=itemBrow->sequence;
                int infCount=itemBrow->count;
                deque<int> changedSeq=changedBrow->sequence;
                int changedCount=changedBrow->count;

                while(jj<infSeq.size()&& kk<changedSeq.size()&&!matchFlag)
                {
                        if(infSeq[jj]==changedSeq[kk])
                        {
                                kk++;
                                if(kk==changedSeq.size())
                                matchFlag=true;
                        }
                        jj++;
                }

                if(matchFlag==true)
```

```
                {
                        oldTemp=itemBrow;
                        changedTemp=changedBrow;
                        itemBrow++;
                        changedBrow++;
                        oldSeq.erase(oldTemp);

                        if(erase2Seq==true)
                                largeToSmall.erase(changedTemp);

                                changedBrow=largeToSmall.begin();

                                compareF=false;
                }
                else
                {
                        if(changedBrow != largeToSmall.end())
                                changedBrow++;
                        else if (itemBrow != oldSeq.end())
                        {
                                itemBrow++;
                                changedBrow=largeToSmall.begin();
                        }
                                else
                                {
                                        compareF=true;
                                }

                }
        }//end while common

          remainedOldSeq=oldSeq;
                return remainedOldSeq;
}
//end of remainedSeq

void PL_RS(list<candidate> combineSeq, double origF,double minT,list<candidate>&PL,list<candidate>&RS)
{

        list<candidate> ::iterator ui;
        struct candidate newCand;
        for( ui = combineSeq.begin(); ui != combineSeq.end(); ui++ )
        {
                deque<int> testS=ui->sequence;
                int testC=ui->count;
                newCand.sequence=testS;
                newCand.count=testC;
                if(testC>=minT && testC<origF)
                {
                        PL.push_back(newCand);
                }
                else if(testC<minT)
                {
                        RS.push_back(newCand);
                }
        }
}

node* updatedWAPtree(ifstream& inFile1,ifstream& inF,ifstream& inFile2, sequence seqLToS,sequence seqLToL,sequence
seqUpedL,node* root1,list<linkheader>&lnkhdr1)
{
        int cid,count;
        sequence::iterator pnt;
        list<linkheader>::iterator plnk=lnkhdr1.begin();
        node* tr;
        node* temp;
        if(seqLToS.size() !=0)//if-seqLToS
        {
                for(pnt=seqLToS.begin(); pnt!=seqLToS.end(); pnt++)
                {
```

```
                        while(plnk->event !=pnt->first)
                                plnk++;

                        //travel WAPtree to del LToS
                        tr=plnk->link;
                        Sub_update(tr,pnt);

                }

        }//end if-seqLToS

        reduceCount(inFile1,root1,seqLToL);

        bool addCount=false;
        bool positionLink=true;

        addBranch(inF,root1,lnkhdr1,seqUpedL,addCount,positionLink);
        return root1;
}

void Sub_update(node* tr,sequence::iterator pnt)
{
        node* temp;
        node* tempparent;

        if (tr->event!=pnt->first)
        {
                tempparent=tr->parent;
                temp=tempparent->parent->lSon;

                Sub_combine(temp,tempparent,pnt);
        }
        else
        {
                if(tr->nextLink!=NULL)
                {
                        node* ptr=tr;
                        Sub_update(tr->nextLink,pnt);
                        tr=ptr;
                        tr->nextLink=NULL;
                }

                if(tr->nextLink==NULL)
                {
                        if(tr->lSon==NULL)
                        {
                                if(tr->rSibling==NULL)
                                {
                                        if(tr->parent->lSon==tr)
                                        {
                                                tr->parent->lSon=NULL;
                                                free(tr);
                                                return;
                                        }
                                        else
                                        {
                                                tr->parent->lSon->rSibling=NULL;
                                                free(tr);
                                                return;
                                        }
                                }
                                else
                                {
                                        temp=tr->parent->lSon;
                                        if(temp->event == tr->event)
                                        {
                                                tr->parent->lSon=tr->rSibling;
                                                free(tr);
                                                return;
                                        }
```

```
                                                    else
                                                    {
                                                            while(temp->rSibling!=tr)
                                                                    temp=temp->rSibling;

                                                            temp->rSibling=tr->rSibling;
                                                            free(tr);
                                                            return;
                                                    }

                                    }
                            }
                            else
                            {
                                    node* ptr=tr;
                                    Sub_update(tr->lSon, pnt);
                                    tr=ptr;
                                    tr->nextLink=NULL;
                            }
                    }
            }

}

void Sub_combine(node* lSon, node* parent,sequence::iterator pnt)
{
            node* temp;
            node * tr;
            temp=lSon;
            tr=parent;


            while(temp->rSibling!=tr)
                    temp=temp->rSibling;

            if (tr->rSibling==NULL)
            {
                    CheckSibling(temp,tr->lSon);
            }
            else
            {
                    if(tr->lSon->event!=temp->event)
                    {
                            if(tr->lSon->event!=tr->rSibling->event)
                            {
                                    temp->rSibling=tr->lSon;
                                    tr->lSon->rSibling=tr->rSibling;
                                    free(tr);
                            }
                            else
                            {
                                    CheckSibling3(tr->lSon, tr->rSibling);
                            }
                    }
                    else
                    {
                            CheckSibling(temp,tr->lSon);
                    }
            }
}

void CheckSibling(node *tempin, node*trin)
{
            node *temp;
            node * tr;
            temp=tempin;
            tr=trin;
            if(temp->event!=tr->event)
            {
                    temp->rSibling=tr;
```

```
                tr->parent=tr->parent->parent;
                free(tr->parent);
        }
        else
        {
                temp->occur=(temp->occur)+(tr->occur);
                if(tr->rSibling==NULL)
                        temp->rSibling=NULL;
                else
                        temp->rSibling=tr->rSibling;

                if(temp->lSon==NULL)
                {
                        if(tr->lSon==NULL)
                        {
                                free(tr->parent);
                                free(tr);
                        }
                        else
                        {
                                temp->lSon=tr->lSon;
                                free(tr->parent);
                                free(tr);
                        }
                }
                else
                {
                        if(tr->lSon==NULL)
                        {
                                free(tr->parent);
                                free(tr);
                        }
                        else
                        {
                                node* temp3=tr->lSon;
                                temp3->parent=temp;

                                free(tr->parent);
                                free(tr);
                                CheckSibling2(temp->lSon,temp3);
                        }
                }

        }
        return;
}//end CheckSibling

void CheckSibling2(node *tempin, node*trin)
{
        node *temp;
        node * tr;
        temp=tempin;
        tr=trin;
        if(temp->event!=tr->event)
        {
                if(temp->rSibling==NULL)
                        temp->rSibling=tr;
                else
                {
                        while(temp->rSibling!=NULL)
                                temp=temp->rSibling;

                        temp->rSibling=tr;
                }

        }
        else
        {
                temp->occur=(temp->occur)+(tr->occur);
                if(tr->rSibling==NULL)
```

```
                    temp->rSibling=NULL;
                else
                    temp->rSibling=tr->rSibling;

                if(temp->lSon==NULL)
                {
                    if(tr->lSon==NULL)
                    {
                        free(tr);
                    }
                    else
                    {
                        temp->lSon=tr->lSon;
                        free(tr);
                    }
                }
                else
                {
                    if(tr->lSon==NULL)
                    {
                        free(tr);
                    }
                    else
                    {
                        node* temp3=tr->lSon;
                        temp3->parent=temp;

                        free(tr);
                        CheckSibling2(temp->lSon,temp3);
                    }
                }

        }
        return;
}//end CheckSibling2


void CheckSibling3(node *trin, node*tempin)
{
        node *temp;
        node * tr;
        temp=tempin;
        tr=trin;
        if(temp->event!=tr->event)
        {
            tr->rSibling=temp;
            tr->parent=tr->parent->parent;
            tr->parent->parent->lSon->rSibling=tr;
            free(tr->parent);
        }
        else
        {
            tr->occur=(tr->occur)+(temp->occur);
            tr->parent=tr->parent->parent;
            tr->parent->parent->lSon->rSibling=tr;

            if(temp->rSibling==NULL)
                tr->rSibling=NULL;
            else
                tr->rSibling=temp->rSibling;

            if(temp->lSon==NULL)
            {
                free(tr->parent);
                free(temp);
            }
            else
            {
                if(tr->lSon==NULL)
                {
```

```
                                        tr->lSon=temp->lSon;
                                        free(tr->parent);
                                        free(temp);
                                }
                                else
                                {
                                        node* temp3=temp->lSon;
                                        node* tr3=tr->lSon;

                                        free(tr->parent);
                                        free(temp);

                                        CheckSibling2(tr3,temp3);
                                }
                        }
                }

                return;
}//end CheckSibling3

void reduceCount(ifstream &inFile,node *start,sequence lToL)
{
        node *Tranversal, *newNode, *Parent;
        int cid,number,event;
        sequence seq=lToL;
        node * root=start;

        if ( !inFile)
        {
                cerr << " File could not be opened\n";
                exit(1);
        }

        while (inFile && !inFile.eof())//while-inFile
        {
                inFile >> cid;
                inFile >> number;

                bool findSeq=false;
                Tranversal = root;

                for(int i=0; i< number; i++)//for-int
                {
                        inFile >> event;
                        if(seq.find(event) != seq.end())//if-seq
                        {
                                findSeq=true;
                                Parent = Tranversal;

                                if( Tranversal->lSon == NULL)
                                {}
                                else
                                {
                                        Tranversal = Tranversal->lSon;
                                        if ( Tranversal->event == event)
                                        {
                                                Tranversal->occur--;
                                        }
                                        else
                                        {
                                                bool find= false;
                                                while(Tranversal->rSibling != NULL && !find )
                                                {
                                                        Tranversal = Tranversal->rSibling;
                                                        if ( Tranversal->event == event)
                                                        {
                                                                Tranversal->occur --;
                                                                find = true;
                                                        }
                                                }
```

```
                                                        if (!find)
                                                        {}
                                                }
                                        }
                                } //end if-seq
                        }//end for-int
                                if(findSeq==true)
                                        start->occur--;
        }

}//end reduceCount

void addBranch(ifstream& inFile, node* root,list<linkheader>&lnkhdr,sequence seqUpedL,bool addC. bool positionL)
{
        linkheader *newLinkHeader;
        node *Tranversal, *newNode, *Parent;
        int cid,number,event;
        sequence seq=seqUpedL;

        if ( !inFile)
        {
                cerr << " File could not be opened\n";
                exit(1);
        }
        while (inFile && !inFile.eof())//while-inFile
        {
                deque<int>inSequence;

                inFile >> cid;
                inFile >> number;

                bool findSeq=false;
                Tranversal = root;

                for(int i=0; i< number; i++)//for-int
                {
                        inFile >> event;
                        inSequence.push_back(event);

                        if(seq.find(event) != seq.end())//if-seq
                        {
                                findSeq=true;
                                Parent = Tranversal;

                                if( Tranversal->lSon == NULL)
                                {
                                        newNode = new node;
                                        newNode->event = event;
                                        newNode->occur = 1;
                                        newNode->lSon = NULL;
                                        newNode->rSibling = NULL;
                                        newNode->nextLink = NULL;
                                        newNode->CountSon = 0;
                                        newNode->parent = Parent;
                                        newNode->pcLength = 0;//Tranversal->pcLength + 1;
                                        newNode->pcCode = NULL;//makeCode(Tranversal->pcLength,Tranversal->pcCode,true);

                                        Tranversal->lSon = newNode;
                                        Tranversal = newNode;
                                }
                                else
                                {
                                        Tranversal = Tranversal->lSon;
                                        if ( Tranversal->event == event)
                                        {
                                                Tranversal->occur++;
                                        }
                                        else
                                        {
                                                bool find= false;
```

```
                                                    while(Tranversal->rSibling != NULL && !find )
                                                    {
                                                            Tranversal = Tranversal->rSibling;
                                                            if ( Tranversal->event == event)
                                                            {
                                                                    Tranversal->occur ++;
                                                                    find = true;
                                                            }
                                                    }
                                                    if (!find)
                                                    {
                                                            newNode = new node;
                                                            newNode->event = event;
                                                            newNode->occur = 1;
                                                            newNode->lSon = NULL;
                                                            newNode->rSibling = NULL;

                                                            newNode->nextLink = NULL;
                                                            newNode->CountSon = 0;
                                                            newNode->parent = Parent;
                                                            newNode->pcLength = 0;
                                                            newNode->pcCode = NULL;

                                                            Tranversal->rSibling = newNode;
                                                            Tranversal = newNode;
                                                            }
                                            }
                                                    }
                                            } //end if-seq
                    }//end for-int
                    if(findSeq==true)
                            root->occur++;
                    if(addC==true)
                    {
                        //accumulating support of candidates of candList in changed part or updated part
                        for(int ii=2; ii<=oldTotalSeq; ii++)
                        {
                            for ( list<candidate>::iterator candBrow = remainedOldSeq[ii].begin(); candBrow !=
                                    remainedOldSeq[ii].end(); candBrow++)
                            {
                                    deque<int> candSequence = candBrow->sequence;
                                    unsigned int j = 0, k = 0;
                                    bool find = false;

                                    while ( j < candSequence.size() && k < inSequence.size() && !find )
                                    {
                                            if ( candSequence[j] == inSequence[k])
                                            {
                                                    j++;
                                                    if ( j == candSequence.size())
                                                        find = true;
                                            }
                                            k++;
                                    }
                                    if ( find )
                                    {
                                            candBrow->count ++;
                                    }
                            }
                        }
                    }
            }//end while-inFile

            if(positionL==true)
            {
                    lnkhdr.clear();
                    for (sequence::iterator i = seq.begin(); i!=seq.end(); i++)
                    {
                            newLinkHeader = new linkheader;
                            newLinkHeader->link = NULL;
```

```
                    newLinkHeader->lastLink = NULL;
                    newLinkHeader->event= i->first;
                    newLinkHeader->occur= i->second;
                    lnkhdr.push_back(*newLinkHeader);
                    free(newLinkHeader);
                }
                cout<<"End of building tree and begin to build linkage..."<<endl;
                positionLinkage(root->lSon,lnkhdr,true);
                cout<<"End of building linkage...\n";
        }
}//end addBranch

void positionLinkage(node *start,list<linkheader>&lnkhdr,bool leftRight)
{
    if (start !=NULL)
    {
        list<linkheader>::iterator lnkBrow = lnkhdr.begin();
        while (lnkBrow->event != start->event && lnkBrow != lnkhdr.end())
            lnkBrow++;

        node *lastLinkage;
        lastLinkage = lnkBrow->lastLink;
        if (lastLinkage == NULL )
        {
                lnkBrow->link = start;
                start->parent->CountSon=start->parent->CountSon+start->occur;
                if(leftRight==true)
                {
                        start->pcLength=start->parent->pcLength+1;
                        start->pcCode=makeCode(start->parent->pcLength,start->parent->pcCode,leftRight);
                }
                else
                {
                        if(start->parent->lSon->rSibling==start)
                        {
                                start->pcLength=start->parent->lSon->pcLength+1;
                                start->pcCode=makeCode(start->parent->lSon->pcLength,start->parent->lSon->pcCode,leftRight);
                        }
                        else
                        {
                                node* temp=start->parent->lSon;
                                while(temp->rSibling!=start)
                                temp=temp->rSibling;
                                start->pcLength=temp->pcLength+1;
                                start->pcCode=makeCode(temp->pcLength,temp->pcCode,leftRight);
                        }
                }
        }
        else
        {
                lastLinkage->nextLink = start;

                start->parent->CountSon=start->parent->CountSon+start->occur;
                if(leftRight==true)
                {
                        start->pcLength=start->parent->pcLength+1;
                        start->pcCode=makeCode(start->parent->pcLength,start->parent->pcCode,leftRight);
                }
                else
                {
                        if(start->parent->lSon->rSibling==start)
                        {
                                start->pcLength=start->parent->lSon->pcLength+1;
                                start->pcCode=makeCode(start->parent->lSon->pcLength,start->parent->lSon->pcCode,leftRight);
                        }
                        else
                        {
                                node* temp=start->parent->lSon;
                                while(temp->rSibling!=start)
                                temp=temp->rSibling;
```

```
                                        start->pcLength=temp->pcLength+1;
                                        start->pcCode=makeCode(temp->pcLength,temp->pcCode,leftRight);
                                }
                        }
                }
                lnkBrow->lastLink = start;

                positionLinkage(start->lSon,lnkhdr,true);
                positionLinkage(start->rSibling,lnkhdr,false);
        }
        else return;
}
```

VITA AUCTORIS

Min Chen was born in Shan Xi, China. He attended Wuhan University of Science & Technology where he obtained a Bachelor degree of Computer Science in 1994. He is currently a candidate for a Master's degree in Computer Science at the University of Windsor and will graduate in the Summer term, 2003.